

UNIVERSITY OF OSLO
Department of Informatics

A MATLAB
Interface to the
GPU

Master's thesis

André Rigland
Brodtkorb

1st May 2007



Abstract

This thesis delves into the field of general purpose computation on graphics processing units (GPGPU). A MATLAB interface for solving numerical linear algebra on the graphics processing unit (GPU), and three algorithms from numerical linear algebra are presented. The algorithms are shown to be faster than the highly efficient ATLAS implementations used in MATLAB. In addition, the interface allows background processing on the GPU, enabling it to be used as a mathematical coprocessor. The computations are shown to be sufficiently accurate, and solving the shallow water equations implicitly is shown where both the CPU and the GPU are both utilized for maximum performance. A comparison of the interface and other high-level languages for GPGPU is also presented.

Contents

Abstract	iii
Contents	v
List of Figures	ix
Listings	xi
Preface	xiii
1. Introduction	1
1.1. Research Question	1
1.2. Organization of thesis	2
2. Background Information	3
2.1. Motivation	3
2.1.1. The GPU as a mathematical processor	3
2.1.2. A high-level interface to the GPU	4
2.2. A brief introduction to the GPU platform	4
2.2.1. The classical rendering pipeline	5
2.2.2. Floating point processing power	7
2.2.3. Different architectures	8
2.2.4. GPGPU	9
2.2.5. Problems with current GPU platforms	13
2.3. MATLAB	16
2.3.1. The MEX API	16
2.3.2. The MATLAB interface implementation	17
3. State-of-the-Art in Numerical Linear Algebra on the GPU	19
3.1. Packing of matrices into textures	19
3.1.1. Full matrices	20

3.1.2. Sparse matrices	22
3.2. GPU optimizations	23
3.3. Numerical linear algebra	24
3.3.1. Matrix multiplication	24
3.3.2. PLU factorization	25
3.3.3. Conjugate gradients	26
4. A Numerical Linear Algebra Interface to the GPU for MATLAB	29
4.1. Algorithms	29
4.1.1. Matrix-matrix multiplication	29
4.1.2. Gauss-Jordan elimination	40
4.1.3. PLU factorization	50
4.2. The GPU as a mathematical coprocessor	57
4.2.1. Nonblocking calls	58
4.2.2. Threading	58
4.2.3. Multi-threading in single-threaded environment	60
4.2.4. Automatically tuned workload distribution	62
5. Application	65
5.1. The shallow water equations	65
5.1.1. Implicit discretization	66
5.1.2. Setting up the linear system	69
5.2. Implementation	70
5.2.1. MATLAB reference implementation	70
5.2.2. MATLAB implementation using the GPU toolbox	70
5.3. Results	70
6. High-level languages	73
6.1. PeakStream	73
6.2. RapidMind	74
6.3. CUDA	74
6.4. Comparison	74
7. Conclusions	79
7.1. Conclusions	79
7.2. Summary of Contributions	80
7.3. Future Research	80
A. PLU factorization on a cluster of GPUs	83
A.1. Introduction	84
A.2. Background	85
A.3. Algorithm	86
A.3.1. Global algorithm	87
A.3.2. Local algorithm	89

A.4. Results	92
A.4.1. Benchmark	92
A.4.2. Analysis	95
A.5. Conclusions and further research	95
A.6. Acknowledgements	96
B. Data Tables	97
C. Shallow water equations source code	101
Bibliography	103

List of Figures

2.1. OpenGL pipeline	6
2.2. CPU and GPU dies.	9
2.3. Reduction operation	14
3.1. Packing of full matrices	21
3.2. Examples of sparse matrices	22
3.3. Packing of sparse matrices	23
4.1. Virtual cube of processors to compute the matrix product . .	31
4.2. Visualization of matrix-matrix multiplication on the GPU . .	34
4.3. Benchmark of full matrix multiplication	38
4.4. Matrix multiplication error	40
4.5. Comparison of norms used to locate the pivot element . . .	45
4.6. The two passes used in Gauss-Jordan elimination	47
4.7. Benchmark of Gauss-Jordan elimination	49
4.8. Gauss-Jordan elimination compared to MATLAB's LU	50
4.9. RREF error	51
4.10. PLU factorization	53
4.11. Benchmark of PLU factorization	56
4.12. PLU factorization error	57
4.13. Deadlock	60
4.14. Design of the MATLAB toolbox	61
4.15. Flowchart of the MATLAB toolbox	62
4.16. Benchmark of the GPU as a coprocessor	63
5.1. Staggered grid	67
5.2. Stencils for solving the shallow water equation implicitly . .	67
5.3. Streamline integration	69
5.4. Solution of the shallow water equations	71
A.1. Parallel PLU factorization	87

A.2. Data send patterns for parallel PLU	88
A.3. Data representation for parallel PLU	90
A.4. Setup of nodes for parallel PLU	92

Listings

2.1. A sample vertex shader	11
2.2. A sample fragment shader	12
2.3. MATLAB function call.	17
2.4. MEX file entry point.	17
3.1. Scalar MAD operation	19
3.2. Vectorized MAD operation	20
4.1. Vector-vector inner product shader	35
4.2. Virtual cube of processors shader	37
4.3. Normalizing shader	48
4.4. Reduction shader	48
4.5. Normalizing shader	55
4.6. Multiplier shader	55
4.7. Background execution in MATLAB	63
6.1. PeakStream example code	75
6.2. RapidMind example code	76
6.3. CUBLAS example code	77
A.1. Example on a deadlock in an MPI-2 program	86
A.2. Setting up row- and column-communicators	91
C.1. MATLAB code to solve the shallow water equations	101

Preface

This masters thesis has been developed at SINTEF ICT, under the GPGPU project [SIN06]. It is approximately twelve months work over a period of one and a half year, as a part of a small research group. The work presented in this thesis has been carried out individually, but is influenced by the thoughts and recommendations of the rest of research group.

It has been a great experience to work with such a project over a long period of time, and it has brought me many valuable lessons and experiences; personally, in programming, in typesetting, and in working with a loosely coupled research group. There have been long days, and a lot of hard work, and I feel proud to present the end result in such a fashion.

The work on this thesis has been executed on two machines, a Dell Optiplex GX620 with a Pentium IV 3.00GHz processor, 2 GB of RAM and an NVIDIA 7800GT graphics card, and a second system with an AMD Opteron 165 processor with 1 GB of RAM and an NVIDIA 7800GT graphics card. Both machines run Microsoft Windows XP SP2. As a consequence, the results and findings in this thesis are only verified for these two hardware setups unless otherwise stated explicitly.

To the reader – This thesis is purposefully divided into several chapters, which should be possible to read by themselves. I have tried to reference previously discussed material when needed, even though there most likely are places lacking these references. Nevertheless, I recommend reading the text sequentially to grasp all concepts most effectively.

For complex and large figures, I have added small diagrams in the margin, where the shaded part correspond to the part of the figure currently being discussed. An example of this is shown on page 6.

Acknowledgements – This thesis would not have been possible, in the current state, without great help from many contributors. I would like to start by thanking my supervisors, Knut-Andreas Lie and Trond Runar Hagen for their encouragement, assistance, and exceptional help in reading of drafts.

It is also very important for me to thank my fellow masters students Lars Moastuen, Hanne Moen, Martin Lilleng Sætra and Trygve Fladby, for

many insightful discussions and a good social environment.

I would also like to thank the following persons and institutions: Guo Wei Ma for reading drafts of this thesis; SINTEF ICT for providing us with a good working environment, and a splendid view of the Oslo fjord; the Department of Informatics at the University of Oslo for supplying us with good hardware and beautiful displays; Jon Mikkelsen Hjelmervik for the idea about the TOP500 project and thoughtful discussions; Johan Seland for GPU advice; Jens T. Thielemann for access to the GPUSIP (EasyShader-Lib for MATLAB) early on; RapidMind for access to the RapidMind Development Platform; PeakStream for access to the PeakStream Platform; everyone on #abelvakt for help with translation to intricate L^AT_EX hacks, especially Igor Rafienko; and lastly, everyone else who have played a part in this thesis.

I also thank my beloved Tine for being incredibly patient and supporting me, even when I have been a *grinch* after long hard days.

Finally, I want to thank my family, especially my mother and father. Their love, continued support and guidance have given me experiences and opportunities few people encounter.

Oslo,
April 2007

André Rigland Brodtkorb

Chapter 1

Introduction

"I'm smarter than the average bear"

— Yogi Bear

This thesis delves into the field of GPGPU, general purpose computation on graphics processing units. The specific area of research is numerical linear algebra using the graphics processing unit (GPU), as the main computational resource.

1.1 Research Question

Even though there exists a lot of research in the field of linear algebra on the GPU [OLG⁺07], there are still some fully or partly unanswered questions. This thesis tries to identify and answer some of these questions. The three main questions posed are:

1. Is it possible to create a toolbox for MATLAB that transparently uses the GPU as a mathematical coprocessor?
2. Is the toolbox accurate enough for use in high-performance applications?
3. Is such an approach competitive, compared with other high-level interfaces to the GPU?

The first question is posed with two things in mind. First, is it actually possible to create a toolbox for MATLAB utilizing the GPU, and second, is it possible to abstract away the intricate details needed to program the GPU? Even though there exists several interfaces to the GPU [OLG⁺05], they all

require some knowledge of the GPU hardware and GPGPU programming paradigms for efficient use.

The next question asks whether such a toolbox is of practical interest. Algorithms on the CPU benefit from double precision floating point arithmetic, whilst only single precision is available on the GPU today. This increases the roundoff errors significantly, possibly leaving our algorithms too inaccurate for practical use.

Finally, the MATLAB toolbox is compared with other commercial and non-commercial interfaces to the GPU. Is the level of abstraction justifiable with respect to these criteria?

1.2 Organization of thesis

The thesis is organized around these main research questions. In the following chapter I motivate the need for such a toolbox, followed by a review the GPU platform, and how to program MATLAB.

Chapter 3 gives an overview of the state-of-the-art of numerical linear algebra on the GPU, and summarizes reported accuracy, stability, and speed for selected algorithms.

After the review of the state-of-the-art, my approach is covered in Chapter 4. First, a selection of algorithms are presented, followed by a presentation of the interface between MATLAB and the GPU. Then, the MATLAB toolbox itself is presented.

The toolbox is used to solve the shallow water equation in Chapter 5, and compared against a reference implementation in Matlab. The toolbox is then compared to other high-level GPGPU languages in Chapter 6.

In Chapter 7 the thesis is summed up, reviewing the most important findings and a set of relevant questions this thesis does not cover.

In addition to the main areas of research, a parallel algorithm for computing the PLU factorization is presented in Appendix A. This is a cooperative project between Martin Lilleng Sætra, Trygve Fladby, and myself. The presented algorithm runs on a cluster of computers to compute the PLU factorization of a matrix, using the GPU as the main computational engine.

Chapter 2

Background Information

“The scout’s motto is founded on my initials, it is: BE PREPARED”

— Lord Baden-Powell

2.1 Motivation

Numerical linear algebra is a topic of great interest. A large number of technical software from search engines [LM04] to games, cryptology and quantum computing [Kni04] use algorithms from numerical linear algebra in some way. Many of these algorithms are computationally expensive and memory demanding. Depending on the application, the time spent solving linear systems can be anything from a couple of CPU seconds to several hundred CPU years. Because linear algebra is such a fundamental part of many algorithms, it often represents a bottleneck. Improving performance will reduce the overall computational time, or enable larger problems to be solved in the same amount of time.

2.1.1 The GPU as a mathematical processor

The massive power of the GPU can be used to solve mathematical problems. The GPU has evolved into a highly specialized processor designed to transform input data, in form of geometry, to pixels on screen. Rendering a large number of screen pixels (typically more than $1280 \times 1024 \approx 1300000$) at least 30 times per second requires a huge amount of processing power. This has made the GPU a far more powerful processor than the CPU, with almost one order of magnitude more floating point processing power [OLG⁺07]. When comparing the price of current high-end CPUs against GPUs, we see that one floating point operation per second (FLOPS)

on the CPU costs over $12\times^1$ more than on the GPU as well, making the GPU a cost-efficient processing unit.

However, this specialized hardware comes at another price. Even though the GPU is a far more powerful processor, the CPU is more flexible. Traditionally, the GPU has to be programmed using a graphics API such as OpenGL [Khr07] and DirectX [Mic07b]. This is useful when rendering graphics, but an obstacle when trying to implement advanced rendering techniques, or non-graphical algorithms. The reason is that the graphics API is predefined to contain only certain functions, thus lacking the ability to compute anything else. Some problems can be mapped to graphical terms, but it can be very cumbersome to program, often yielding far from optimal results.

Recently, however, graphics vendors have begun to expose more of the inner workings of processors, enabling us to run more general rendering code on the GPU. This allows us to perform much more complex renderings, and compute complex non-graphical problems with less difficulty. As a result, we can harvest the massive power of the GPU for selected algorithms, e.g., in numerical linear algebra. Many algorithms implemented on the GPU have been shown to be faster than highly optimized CPU code [KW03, GGHM05, MWHL06, Mor03, GLGM06].

2.1.2 A high-level interface to the GPU

Utilizing the GPU as a computational resource can efficiently speed up existing applications with a modest price tag. Existing solutions, such as PeakStream [Pea06], RapidMind [MD06] and CUDA [NVI07b] require insight into the GPU programming model, and offer interfaces to C/C++ only. Giving access to the power of the GPU via a high-level language, such as MATLAB, lowers the bar for entering the GPU domain, and abstracts away all of the intricate details needed to program the GPU.

In addition, the use of the GPU as a computational resource most often excludes the use of the CPU. Very few, if any, scientific papers in this field, mention this point, focusing mainly on the strengths of the GPU. However, many problems can benefit from using the GPU *and* the CPU simultaneously, thus increasing the overall speed significantly.

2.2 A brief introduction to the GPU platform

The GPU is designed to compute the screen image displayed on modern computer monitors. The screen image is computed from geometry in two or three dimensions, which is flattened and sampled at regular intervals into discrete screen pixels.

¹Prices are from a Norwegian web-shop 2007-04-23

The input geometry to the GPU is given as homogeneous 3D coordinates, $[x, y, z, w]$. Homogeneous coordinates include the fourth dimension, w , enabling affine transformations, e.g., translation, rotation and scaling, to be represented as matrix multiplications. The w dimension can be viewed as a scaling of the coordinate system, and is scaled to 1 by the GPU in the perspective division.

The technique for computing the color of each pixel uses the RGB color model to represent a real color. A real color can contain an infinite number of different color frequencies, but the RGB color model decomposes the color into the three colors red, green and blue, resulting in a color that appears similar to the original to the human eye. In addition to the three color channels in the RGB color model, a fourth color channel is used by GPUs: the alpha channel. The alpha channel is used to represent opacity, yielding the possibility of transparent objects layered on top of each other. This color model is referred to as the RGBA color model.

In order to efficiently compute the four channels in the RGBA color model, and the four coordinates $[x, y, z, w]$ of the input geometry, many GPUs have traditionally been designed to compute them in parallel as one vectorized operation. In effect, each processor on the GPU can compute four multiply and add (MAD) operations per clock cycle.

Traditionally, computing the color and position has been fixed in hardware, but newer graphics cards can be programmed in a more general fashion. This enables not only complex rendering algorithms, but also the possibility to compute the solution to other, more general problems [OLG⁺07]. Programming of the GPU for non-graphics use is often referred to as general purpose computation on graphics processing units (GPGPU), and has become a field of great interest. The reason is the vast processing capabilities of the GPU, combined with a very modest price tag.

2.2.1 The classical rendering pipeline

The process in which screen pixels are computed from input geometry is often referred to as the rendering pipeline. There have been several interfaces to the GPU, the two main ones today being OpenGL [Khr07] and DirectX [Mic07b]. The two APIs are more or less equivalent with only minor differences. I have used OpenGL in this thesis, and will briefly review the rendering process as seen from an OpenGL point of view.

OpenGL is implemented as a state machine [SWND05]. A state machine is a programming technique where the computation of data depends on the current state. If you for example have a lighted scene, the rendering of the scene will only include lighting if the machine is in the proper state, i.e., lighting enabled. When the lighting state is set, all data is processed with lighting enabled until it is explicitly turned off.

In addition to being a state machine, OpenGL is modeled as a pipeline,

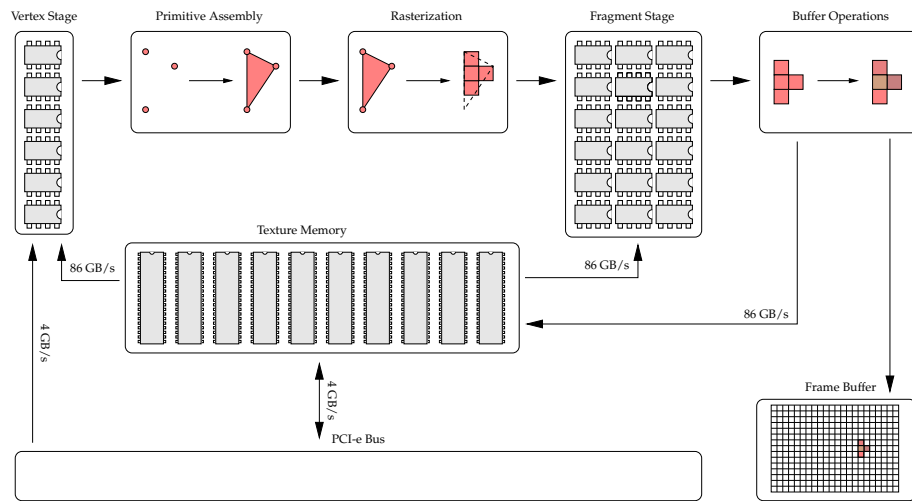
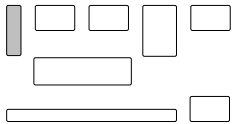
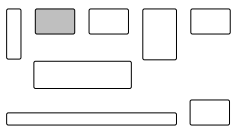


Figure 2.1: The OpenGL pipeline. The internal bandwidth on the GPU, 86GB/s is on the NVIDIA GeForce 8800 series GPUs. The PCI-e bandwidth, 4GB/s, is for the PCI-e 16× bus.

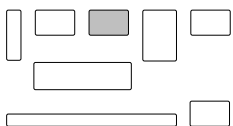
as shown in Figure 2.1. All output pixels are computed from input data that has to pass through all the five main stages of the pipeline before the final color is determined.



Vertex stage In this stage, all values belonging to a vertex are computed and set. A vertex is a node in a geometry; e.g., a corner in a triangle. Each vertex is usually transformed into view coordinates, and transformation of attributes, such as color, texture coordinates and normals, is also performed here. In Goraud shading, lighting calculations are executed here, using the vertex normal to perform per-vertex lighting.



Primitive Assembly The primitive assembly stage is where all the vertices are assembled into primitives such as triangles and quadrilaterals according to the current state. When all the vertices have been assembled, they are clipped and tessellated so that no primitives exist outside our field of view.

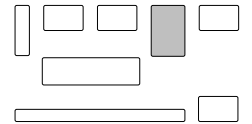


Rasterization stage In the rasterization stage, the scene is sampled at regular intervals into fragments. Fragments are meta-pixels that have not yet been written to screen. The color, depth, and other attributes of the fragments are set using barycentric interpolation [SWND05]. Barycentric interpolation interpolates the values given at each vertex in the primitive to the wanted internal point.

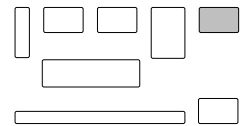
The rasterization stage is deterministic, so that it is well defined which

primitive a fragment belongs to. This is important for fragments that cover the intersection between two primitives. Without this determinism, such fragments could end up constantly changing color, which of course is an unwanted artifact.

Fragment stage Up until now, fragments have only been assigned a color and other attributes that have been interpolated from neighbouring vertices. To create realistic renderings, textures are added in this stage. The texture can for example be blended with the underlying color, or substitute the color completely. We can also perform other calculations here, such as per-fragment lighting. Phong shading, for instance, uses interpolated normals to calculate the fragment color, which gives a smoother and more realistic shading than Gouraud shading.



Buffer operations In the buffer operations stage, fragments can be blended, written, or discarded according to the state of OpenGL and the value of buffers such as the depth buffer or stencil buffer. When a fragment has passed all tests, and been blended with over- or underlying fragments, it is finally written to the framebuffer.



This is a very simplified model, but it captures the stages that can be exploited for GPGPU, mainly the vertex and fragment stage [OLG⁺05]. In addition to these two stages, Shader Model 4.0 [Bly06] cards implement a programmable geometry shader that replaces the primitive assembly stage in the classical pipeline [Mic07a].

2.2.2 Floating point processing power

The development of CPUs has followed Moore's law closely, doubling the number of transistors every 24th month [Wik07b]. Traditionally, the clock speed and arithmetic capacity of the processors have followed similar curves. Recently, however, the processor speeds have not been able to keep up [TSV05]. Physical restraints, among other heat problems, have disrupted the increase in frequency, leading to major architectural changes such as multi-core processors [TSV05]. The GPU, on the other hand, has experienced an even faster growth curve than that of the CPU, even sustaining its growth rate when the CPU growth rate has stalled. The arithmetic capacity of the GPU has doubled every 6 months, outperforming the CPU by far [OLG⁺05].

As new GPU and CPU models appear on the market, the gap in processing capacity increases, with current high-end consumer level GPUs capable of ~ 400 GFLOPS [Neo07], while high-end CPUs are capable of ~ 90 GFLOPS [Neo07].

2.2.3 Different architectures

Even though the GPU is a far more powerful processor in terms of GFLOPS, the processor speeds do not tell the full tale by themselves. The GPU and the CPU employ two entirely different operating modes, addressing different needs. The CPU has complex logic for branch prediction, cache control and instruction pipelining, while the GPU has little of this logic. It is simply not needed for the specialized operations used to compute screen images, and forces the programmer to think in graphical terms where not all operations are legal.

The CPU is optimized for serial execution of instructions; perfect for tasks such as word processing, where every character is entered and processed sequentially. This processor type is called a single instruction, single data (SISD) processor, where *one* instruction is applied to *one* data value per clock cycle.

The GPU on the other hand, is not only one processor, but several operating in parallel. The latest generation of graphics cards utilize a unified shader architecture, in which each processor can act as a vertex, geometry or fragment shader [Mic07a]. Traditionally, however, there have only been designated processors for the vertex and fragment stage, with the geometry stage implemented with fixed functionality.

In the vertex stage, the processors compute in parallel as multiple instruction, multiple data (MIMD) processors. The MIMD architecture is optimized to execute different instructions for different data, i.e., *multiple* instructions over *multiple* data values.

The fragment processors, however, operates as single instruction, multiple data (SIMD) processors. In comparison to the SISD architecture, which is optimized for sequential code, the SIMD architecture of the fragment processor is optimized for computing the same instruction over a large set of data, i.e., *one* instruction is applied to *multiple* data values. The SIMD architecture is very efficient for a group of parallel problems, but not all. It lacks efficiency when different instructions are needed for different data. Computing the output color of pixels, however, is an example of a problem for which fragment processors are very powerful.

Graphics cards have a relatively fast interface to the CPU via the PCI Express expansion slots, with a theoretical bandwidth of 4 GB/s full duplex (4GB/s simultaneous read and write) [Mic04]. Even more important is the internal bandwidth on the graphics card. Current high-end graphics cards have a theoretical bandwidth of 86.4GB/s [NVI07c] to DDR3 texture memory. High-end CPUs, on the other hand, typically have a theoretical bandwidth of only 8.5 GB/s to main memory [Bes04].

Even though the GPU has far more efficient access to memory, the CPU has more on-chip cache. A high-end CPU typically has two or four MB of cache. Since the architecture of the GPU is closed [JS05], the exact size

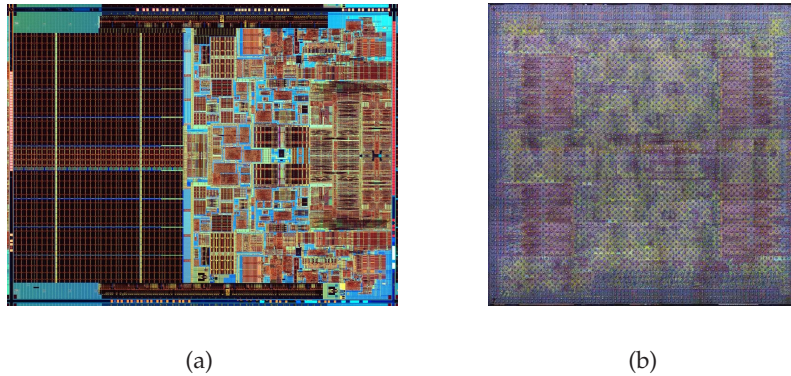


Figure 2.2: Colored photographs of processors, showing layout of one CPU and one GPU. The images are not to scale: (a) Intel Core Duo (Extreme) with 291 million transistors. (b) NVIDIA GeForce 8800 series with 681 million transistors.

and organization of cache is not publicly available. Nevertheless, Govindaraju et al. points to 8×8 as a probable size on the NVIDIA GeForce 7800GTX [GLGM06]. This cache size represents the two-dimensional 8×8 neighbourhood of a single element, in contrast to the regular one-dimensional neighbourhood used for CPU cache.

Because the CPU has such a large on-chip cache, many of the transistors in each processor are used for this purpose. Figure 2.2(a) shows a colored photograph of a CPU die, where the large brown part on the left hand side is the cache. The right hand side consists of two processing cores, where many of the transistors are used for memory management, logical operations (e.g., branch prediction) and instruction pipelining. In total, only a small fraction of the 291 million transistors are used for arithmetic operations. The GPU, on the other hand, does not have a massive on-chip cache or complicated logic. Here, the majority of the transistors are used for arithmetic operations. A colored photograph of a GPU is shown in Figure 2.2(b), showing no large cache areas. In total, the majority of the 681 million transistors are used for floating-point arithmetic.

2.2.4 GPGPU

The two programmable stages in the graphics pipeline, namely the vertex and fragment stage, are the ones we exploit for GPGPU. Of the two, the fragment stage is the most powerful by far. This is mostly because the gaming industry is the main drive behind the development of consumer-level graphics. In any given scene in today's hottest games, there are far more op-

erations computed in the fragment processor than in the vertex processor. Because of this, most of the processing power lies in the fragment stage. In current high-end graphics adapters, there are 8 vertex, and 48 fragment processors (ATI Radeon X1900 XTX) [Adv07]. Shader Model 4.0 cards, however, implement a unified shader architecture [Mic07a], where the distinction between fragment and vertex shaders is eliminated. The NVIDIA GeForce 8800 GTX, for example, has 128 unified shader cores [NVI07c].

Terminology and algorithms

To utilize the fragment shader for general computation, we have to render graphical primitives representing our problem. In addition to formulating the problem as primitives, we have to formulate it for efficient execution on the specialized hardware. As with the CPU community, the GPGPU community has developed its own set of standard algorithms and terms that arise in many applications. In the specific field of numerical linear algebra on the GPU, I utilize a subset of these terms and algorithms. The following non-exhaustive list gives an overview of the most important terms and algorithms used in this thesis:

Gather and scatter – Gather and scatter are memory operations. Gather can be viewed as the C statement

```
k = data[i];
```

where data is *gathered*, or read, from a specific memory address. Scatter is the exact opposite, where data is *scattered*, or written, to a specific address in memory:

```
data[i] = k;
```

The GPU is not directly capable of scatter operations, except with the CUDA interface to the NVIDIA GeForce 8800 series GPUs. It can be emulated, however, using the vertex processor to move vertices to the correct memory address (framebuffer position), and the fragment processor to write the correct value.

Shader – A shader is an ambiguous term in the field of GPGPU. It refers to both the processor on which code is executed, e.g., the vertex shader unit, but can also refer to the code being executed, e.g., vertex shader program. The distinction is context sensitive, and almost always easy to make.

Programming a shader is somewhat different from programming regular CPU code; there are a few details that are far from natural, seen from a CPU point of view. There are several languages available for writing shaders, e.g., OpenGL Shading Language (GLSL) [KBR06], High Level Shader Language (HLSL) [Mic07c] and C for Graphics (Cg) [NVI07a], but the one

Listing 2.1: A sample vertex shader

```
1 void main() {  
    gl_Position = ftransform();  
    gl_TexCoord[0] = gl_MultiTexCoord0;  
}
```

used in this thesis is GLSL, the OpenGL Shading Language. The other languages are similar, but I will not discuss them further.

The main primitive data-type in shaders is float. In addition to the float data-type, there is also support for vectors (`vec2`, `vec3`, `vec4`), and matrices (`mat2`, `mat3`, `mat4`). These data-types can have three different qualifiers, `const`, `uniform`, or `varying`. A `const` variable is a compile-time constant, while a `uniform` variable is set from the CPU at runtime. The last qualifier, `varying`, denotes that the variable is set in the vertex shader, interpolated in the rasterizer, and accessible from the fragment shader.

Listing 2.1 shows a typical vertex shader. The shader simply computes the same result that would have been computed by the fixed function pipeline, transferring a vertex from world coordinates to view coordinates. A typical use of the vertex shader is to alter the height of vertices representing water. This is done by looking up the water height in a texture, and setting the vertex height appropriately. The use of vertex shaders in GPGPU is not ubiquitous, but there are examples of use, e.g., for scatter operations [OLG⁺05].

Fragment shaders, on the other hand, are more often used in GPGPU. Listing 2.2 shows an example fragment shader, where the variables `A` and `B` are textures. To access one element in the texture, the function `texture2D` is used with a texture and coordinate as input. The coordinate that is looked up is the texture-coordinate set by the vertex shader (see Listing 2.1).

Then, we set the output color of the fragment by adding together the two colors we looked up. The observant reader will notice that we have used `rgba` at the end of `gl_FragColor`, and `rrgg` and `bbaa` on the two variables `frag1` and `frag2`. These are not typos, but exemplify two important concepts when programming shaders. This syntax is a way of indexing the four elements of the vectors. `rgba` is a perturbation of the four colors in the RGBA color model, rearranging them as red, blue, green and alpha. This is referred to as *swizzling*. The other concept is called *smearing*, and is the replication of elements, i.e., `rrgg` replicates the red and green channels.

Operators in GLSL are executed element-wise. This has the effect that in the sample shader code, the red channel of `gl_FragColor` is computed as the sum of the red channel of `frag1` and the blue channel of `frag2`.

Listing 2.2: A sample fragment shader

```
1 uniform sampler2D A;
  uniform sampler2D B;

  void main() {
5   vec4 frag1 = texture2D(A, gl_TexCoord[0].xy);
     vec4 frag2 = texture2D(B, gl_TexCoord[0].xy);

     gl_FragColor.rbga = frag1.rrgg + frag2.bbbaa;
  }
```

Multi-pass rendering – A multi-pass rendering algorithm is an algorithm that renders the scene multiple times to produce the final output. An example is if you want to render a custom-shaded scene. The ambiently lit scene can be rendered first, with consecutive passes rendering shadows, highlights, etc. The use of multi-pass rendering is very common in the field of GPGPU, where one example is the ping-pong technique described in the next paragraph.

Ping-ponging between buffers – OpenGL explicitly defines the result of writing to the buffer you are reading from as undefined [GGHM05]. This is because the sequence of computation is undefined, i.e., you cannot know which fragments are computed first. Tests on the hardware used in this thesis, however, reveal that that writing to the exact same texel as the one you are reading from works. Nevertheless, it should be avoided to ensure portability of the application.

Being unable to write to the buffer one is reading from is a huge problem for many algorithms such as forward substitution. In forward substitution, you would normally write to the same array. Because reading and writing to the same buffer is undefined in OpenGL, a technique known as ping-ponging is employed. Instead of using just one array, you duplicate the array, and alternate between reading from and writing to the two buffers. The technique is similar to the swapping of time-step arrays in numerical algorithms used to solve many ODEs and PDEs.

For example, in pass one, buffer *A* is input, and buffer *B* output. When the rendering pass is complete, you simply swap the role of the two buffers, so that buffer *B* is input and *A* is output. In OpenGL this means you have to flush the pipeline between passes, but it is the only way of writing to the same virtual array you are reading from.

Reduction operator – A reduction operator is a prime example of using the ping-pong technique. The reduction of a large set of data to a smaller

dataset, or more often, a single scalar, is needed in many algorithms. Examples are finding the maximum element in an array, computing the sum of a vector, or computing the condition number of a matrix.

The process is quite similar to creating MIP maps². The reduction is computed using multiple passes as shown in Figure 2.3. When going from matrix A to B, we read from the original buffer, `orig`, writing to a new buffer, `front`. When going from B to C, we might not want to overwrite the contents of matrix A. Thus, we write to a new buffer again, `back`, reading from buffer `front`. When going from matrix C to D, the roles are reversed, and we read from buffer `front` and write to `back`.

The input array is often much larger than the 8×8 example in Figure 2.3, requiring several more ping-pong passes. The number of passes, n , required to reduce the stream into a single element can be computed by

$$n = \frac{\log(p)}{\log(q)}, \quad (2.1)$$

where p is the number of rows and columns in the array, and q is the factor we are reducing the matrix with per pass.

2.2.5 Problems with current GPU platforms

Because the GPU is so specialized, and generally has to be accessed via a graphics API³, it lacks some of the functionality we take for granted on the CPU. Some of the limitations have been solved as described in the previous subsection, but there are still problems which have no apparent solution.

Lack of double precision and IEEE floating point – Because the GPU is mainly used for rendering, single precision float has been sufficient for a long time to represent colors, texture coordinates and vertex positions. But with the emergence of GPGPU as a field of research comes the need for double precision. Many algorithms in scientific computing are sensitive to rounding errors, and need double precision to be of use. There has been research in the field of emulating double precision on the GPU [Str02, GST05, GST07], but general techniques are costly, and not always applicable. NVIDIA has fortunately revealed that their next generation of GPUs will implement double precision capability, but at the price of half the computational power compared to single precision.

²MIP maps are images that represent the same image at different resolutions. Typical use includes texturing of objects. When the object is far away, it is sufficient to use the low-resolution image, and switch to the higher resolution image when the object is close to the camera.

³The GeForce 8800 chips from NVIDIA have an API called CUDA which can access the hardware without going through a graphics API.

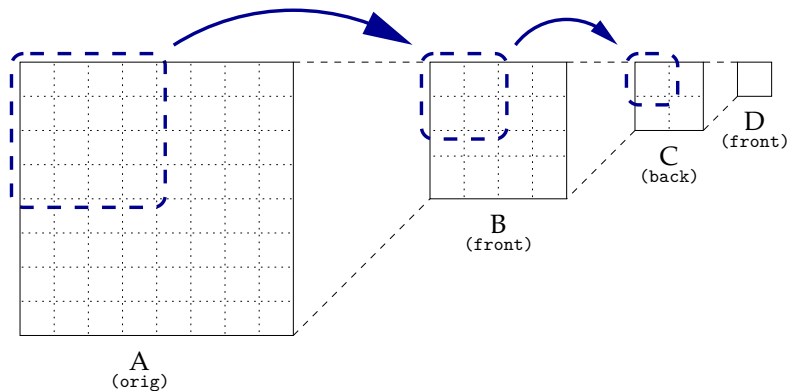


Figure 2.3: A typical reduction of a stream. We start with all the elements in array A, which then are reduced into the smaller array B. This is done by drawing a quad covering array B, with texture coordinates covering array A. In effect, each fragment in array B has the texture coordinates covering four elements in array A. The sum, max, min or user defined operation then is executed on the four elements from array A, and the result is written to the fragment in array B. This process is repeated until the wanted output-size is reached.

The available single precision on GPUs is not IEEE-754 conformant either. The implementation of certain arithmetic operations, as well as handling of NaN, INF, etc. may deviate from the standard causing undesired results [FSC06, DD06].

As a side-note, it should be mentioned that ATI cards have implemented hardware support for fused multiply-add instructions [GPG06]. These are often used to increase precision on single precision hardware, as they only impose one rounding-error as opposed to two rounding-errors which would normally occur. NVIDIA chips have been reported to *not* have the same capability [GPG06], but the technical specification for the newest generation GPUs states that a fused mad operation is implemented [NVI07b].

Missing hardware implementation – The OpenGL standard defines what functionality is necessary to be OpenGL compliant. But it is up to the hardware vendors to actually implement the functionality in hardware. The first functions to be implemented are those used by the hottest games available. With every new driver release, improved functionality is added as well as bug fixes and speed improvements. Functions that may appear obscure from a graphics point-of-view, however, are seldom implemented correctly. To be compliant, they do not cause an error either. They simply do nothing. Some functions are also only partly implemented, working only for specific inputs or states.

It should be noted that when using the fixed function pipeline, one rarely encounters these errors. OpenGL does not specify that the functionality must be implemented in hardware, so many functions are executed on the CPU and can be quite slow. Microsoft Shader Model [Bly06], however, does specify what needs to be implemented in hardware for compliancy. But Microsoft Shader Model naturally does not specify OpenGL requirements. Writing shaders reveals a lot of lacking OpenGL functionality in the hardware. It can be tricky to find these errors, considering that there are no good debuggers available [OLG⁺05]. There does not seem to be any record of these defective functions either (even though some limitations with e.g., texture formats are documented for certain graphics cards [NVI05]). The following list, however, exemplifies the type of difficulties one most likely will encounter when programming the GPU:

- for-loops in shader code are truncated to be maximum 256 loops long without casting an error [JS05]. However, using several consecutive for-loops, each of length less than or equal to 256, achieves the wanted result.
- Using vertex texture fetch requires special texture formats, and the use of normalized texture coordinates [NVI05].

Branching – Branching can be particularly expensive on the GPU because of the way it is implemented. Because of the SIMD architecture, many processors will execute the same instruction in parallel. This means that if one processor branches differently from the others, all processors have to execute both sides of the branch. This is very costly, and should be avoided, or moved to the CPU if possible. The vertex processor, however, uses the MIMD architecture, where branching is less expensive. A lot of improvements in this field have also been presented by the newest hardware generations, where the size of output computed by a group of processors has decreased in size. Nevertheless, one has to consider branching as a possibly expensive operation.

MAD to texture fetch ratio – Even though the GPU has fast access to RAM, it lacks the highly efficient cache found on CPUs. The size of the GPU cache is far smaller than that of the CPU as mentioned previously. Because of the relatively modest cache size, the GPU has to access RAM memory far more often than the CPU even for cache coherent algorithms. The cache on the GPU is also several times slower than what is found on the CPU [FSH04]. To prevent the GPU from idling while data is fetched from cache and RAM, it should be used to perform computations simultaneously. For example, eight MAD instructions per fetched float was reported as the ideal ratio for recent GPUs [FSH04, SDK05].

2.3 MATLAB

MATLAB is a highly tuned mathematical suite. It can operate on matrices and vectors, and visualize the results with very little code in an interactive development environment. The abstraction from the hardware is large, enabling mathematical syntax which is easy to write, maintain and understand. This combination has made it a standard tool for scientists and engineers all over the world.

MATLAB uses the ATLAS [WD98b], LAPACK [WD98a], and BLAS [LHKK79] libraries for its numerical linear algebra algorithms [The06, Mol00]. The mentioned libraries are highly efficient, and the MATLAB interface to them is regarded as highly efficient too [MY02]. Even though MATLAB is highly efficient, the algorithms implemented in the libraries are both computationally and memory heavy. Utilizing the GPU as a coprocessor for these algorithms will possibly enable us to solve the same systems faster than previously possible.

2.3.1 The MEX API

MATLAB is designed to support executables written in C or Fortran. These executables are known as MEX-files, where MEX stands for MATLAB Executable. The MEX API is available by including special MATLAB header files in C. I will not mention the Fortran API further, but concentrate on the API to C, which I utilize in my approach.

The header files define MATLAB specific functions for many built-in C functions such as `malloc` (memory allocation), `free` (memory deallocation) and `printf` (print to standard out). These functions have names such as `mxFree`, `mxMalloc` and `mexPrintf`, respectively, and are divided into four main groups. All functions with `mat` as prefix operate with MATLAB's `mat` files. This is used to read and write matrices to and from disk. The functions that start with `mx`, operate on MATLAB multidimensional arrays, `mxArray`. These functions read and write to and from system memory (RAM). The third group has the prefix `mex` for all functions. These functions deal with interaction between MATLAB and the MEX files. The final group uses the prefix `eng`, and enables us to use the MATLAB engine in other programs.

A typical MATLAB call is shown in Listing 2.3. Here you have several input arguments, (α, β, \dots) , and a vector of output arguments, $[a, b, \dots]$. All arguments are in general matrices. Scalars are simply the special case of a 1×1 matrix, and vectors are the special cases of $1 \times n$ or $n \times 1$ matrices. The input and output arguments are passed to the MEX file via the `mexFunction` shown in Listing 2.4. The `mexFunction` is the entry point for MATLAB. Every time MATLAB calls the library, this function is executed with the four arguments `nlhs`, `plhs`, `nrhs` and `prhs`. Here `nlhs` is the *number of*

Listing 2.3: MATLAB function call.

```
[a, b, ...] = func( $\alpha$ ,  $\beta$ , ...);
```

Listing 2.4: MEX file entry point.

```
void mexFunction(int nlhs, mxArray* plhs[],
                 int nrhs, const mxArray* prhs[]){
    //Function body
}
```

left hand side arguments, *plhs* is an array of pointers to the left hand side arguments, and likewise for the right hand side of the expression. If we take Listing 2.3 as a starting point, and regard only the arguments *a*, *b*, α and β , we get the following situation:

nlhs is two since the number of output arguments is two.

plhs is a pointer to an *mxArray* of size two. The first is a pointer to *a*, and the second a pointer to *b*.

nrhs is two as well, since this is the number of input arguments.

prhs is a pointer to another *mxArray* of size *nrhs*. The first item corresponds to α and the second to β .

This way of handling arguments enables very neat code for mathematical functions. The singular value decomposition can for example be called as $[U, S, V] = \text{svd}(X)$, which models the mathematical notation $USV^T = X$ very well. But the model also adds another layer of complexity when writing a MEX file. The input and output arguments have to be tested and set differently for different arguments. For example, the function *svd* can also be called simply as $S = \text{svd}(X)$ where *S* is a diagonal-matrix with the singular values of *X* on the diagonal. It is therefore often useful to be able to vary the number and/or type of input/output arguments as well. For example if *X* is a sparse or single precision matrix type, *S* should also be a sparse or single precision matrix, respectively.

2.3.2 The MATLAB interface implementation

MATLAB supports classes and operator overloading for user-defined classes. This enables us to implement a very elegant way of communicating with C. By adding a directory to MATLAB's path with a name starting with a commercial at, e.g., @baz, MATLAB recognizes all files within that directory as belonging to that class, i.e., the baz class. So when MATLAB encounters a function called with the baz class as argument, it first searches for an m-file

within the @baz directory with the name of the function. If you, for example, execute `foo(bar)`, where `bar` is of class `baz`, MATLAB would try to run the file `@baz/foo.m` with `bar` as the argument.

There are also some predefined functions that MATLAB uses for special operators. Using the example that `bar` and `qux` are of class `baz`, the following non-exhaustive list of functions have these respective purposes:

`display` The `display` function prints a text representation of the object on screen. MATLAB runs this file when you execute

```
> bar
```

{`plus`, `minus`, [`m`] `times`, [`m`] `ldivide`, [`m`] `rdivide`, etc.} The operators `+`, `-`, `*`, `\`, `/`, etc. can all be overloaded in MATLAB, but the functions are actually called `plus`, `minus`, etc.

For matrices, there are two kind of operators. Element-wise operators, and matrix operators. The matrix operators (where it makes sense) have `m` as a prefix, i.e., `mtimes` which is matrix multiplication. The corresponding element-wise operator `times` simply computes element by element multiplication. In MATLAB notation, you specify that you want element-wise operations by prefixing the operator by a period, i.e., `.*` for element wise multiplication.

The MATLAB call

```
> bar * qux
```

will execute the function in the file `@baz/mtimes.m` with `bar` and `qux` as input arguments.

{`single`, `double`, `int`, etc.} These are functions that convert the data into the respective classes, i.e., calling

```
> single(bar)
```

returns an `mxArray` consisting of single-precision elements. This becomes particularly useful when dealing with custom classes that store the data differently.

Chapter 3

State-of-the-Art in Numerical Linear Algebra on the GPU

“If I have seen further it is by standing on ye shoulders of Giants”

— Sir Isaac Newton

There has been a lot of research in the field of numerical linear algebra on the GPU, going back to when only the fixed function pipeline was available [LM01]. Since then, several new generations of GPUs have emerged on the market, each new generation being more general, and more powerful. As the graphics cards have become more and more general, more advanced algorithms have become possible to implement on the GPU. In this part of the thesis, I examine the state-of-the-art of numerical linear algebra on the GPU.

3.1 Packing of matrices into textures

As mentioned in Chapter 2, the GPU can calculate one multiply and add (MAD) instruction per clock cycle on a full four-long vector. As an effect, the code samples in Listings 3.1 and 3.2 take the same time to compute. It should be mentioned, however, that the GeForce 8800 GPU series from NVIDIA are implemented using scalar arithmetic, where the code in Listing 3.1 would run faster than the code in Listing 3.2.

This thesis concentrates on the architecture previous to the NVIDIA Ge-

Listing 3.1: Scalar MAD operation

```
d.r = a.r * b.r + c.r;
```

Listing 3.2: Vectorized MAD operation

```
d.rgb = a.rgb * b.rgb + c.rgb;
```

Force 8800 GPU series, where it is vital to organize the data in four-long vectors. How we pack the data we compute on, not only influences the use of arithmetic units, but also the number of cache hits and other important aspects. This makes packing of data in an efficient manner a very important part of algorithms on the GPU. It is not trivial, or perhaps even possible, to design a generic packing algorithm that works well for most applications. The distinct differences in memory access patterns for different problems makes it impossible to create a perfect data structure. Nevertheless, several good packing algorithms have been presented, such as quad-trees, stacks, and queues [OLG⁺05, LSK⁺06]. They all address problems with specific data access patterns and needs.

Many algorithms in numerical linear algebra do not require such exotic data-structures, but still need to pack the data effectively. The data in linear algebra is most often matrices, full, structured or sparse. To pack these matrices effectively, we need to use all the knowledge we have of them.

3.1.1 Full matrices

Single component texture – The naïve approach to storing a matrix on the GPU is simply to transfer it as a single component texture. Figure 3.1(a) shows this approach, where the data is stored in one of the four color channels. This naïve approach enables the use of textures in a similar fashion to arrays on the CPU, but only utilizes 25% of the computational power.



Four-by-one packing – Moravánszky [Mor03] presented a way to utilize the full potential of the GPU in 2003. The algorithm simply pads the matrix with zeros before transferring it to the GPU, as shown in Figure 3.1(b). When the data is padded and transferred to the GPU, four-by-one or one-by-four sub-matrices are effectively packed into each fragment. If the matrix size is divisible by four in the packing dimension, no padding is needed, nor added. The pros of this algorithm is that it is very cache friendly when accessing data column- or row-wise, depending on whether four sequential items in one column or one row are packed. It is also relatively easy and intuitive to use, and the padding does not affect any algorithms such as matrix multiplication and matrix solvers.



A good example of where this packing will achieve high efficiency is full matrix-matrix multiplication. In matrix multiplication, one element in the output matrix is the inner product of a column- and a row-vector from

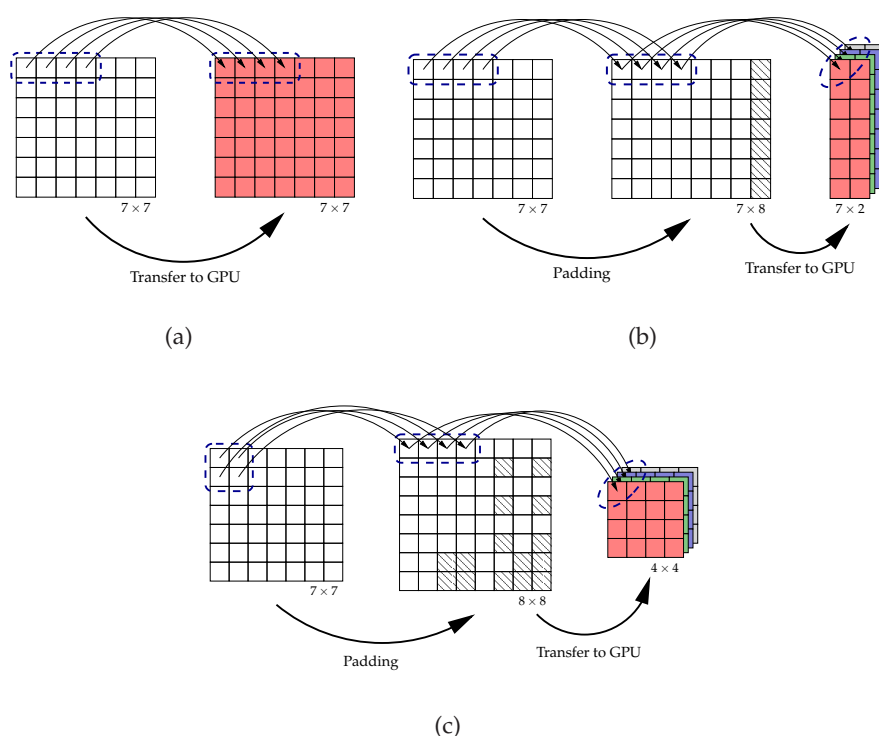


Figure 3.1: Different packing schemes when transferring matrices to the GPU: (a) The simplest packing algorithm which only uses one of the color channels. (b) Four-by-one packing algorithm as proposed by Moravánszky [Mor03]. (c) Two-by-two packing algorithm as proposed by Hall et al. [HCH03].

the input matrices. Packing the two input matrices as one-by-four and four-by-one respectively will increase cache hits.

Two-by-two packing – Another approach to packing a full matrix is the two-by-two packing algorithm first published by Hall et al. [HCH03]. In this packing scheme, the data is packed into small two-by-two sub-matrices. Each fragment contains a small two-by-two sub-matrix, which then is cache friendly both row- and column-wise. But it is more work to pack the data, since it requires more restructuring than the four-by-one scheme. The trade-off, however, is that the packing scheme is efficient when used in different algorithms, as it increases cache locality in both columns and rows. A schematic of the algorithm is shown in Figure 3.1(c).



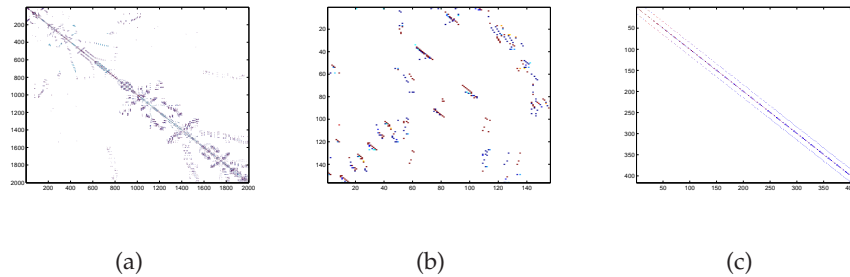


Figure 3.2: Examples of different kinds of sparse matrices. The matrices are part of the University of Florida Sparse Matrix Collection [Dav07a]. The text in parenthesis refers to the id of the corresponding matrix in the collection: (a) A sparse symmetric matrix arising in a computational fluid flow problem (HB/bcsstk13). (b) An unstructured matrix arising in the simulation of a chemical process (HB/west0156). (c) A banded matrix arising in an electromagnetic problem (Bai/mhdb416).

3.1.2 Sparse matrices

There exists several different kinds of sparse matrices [Dav07b]. Because sparse matrices often have few nonzero entries, we must find effective packing algorithms for them on the GPU. This is a difficult task, since representing sparse matrices even on the CPU is nontrivial. There are a lot of different types of sparse matrices, e.g., symmetric and unstructured sparse matrices (see Figure 3.2), all of which have different packing needs. In scientific computing, however, we generally classify sparse matrices into two main categories, banded and general sparse.

Diagonal-vectors – Banded matrices often occur in scientific computing when solving PDEs using a finite-difference or finite-element approach, and are therefore of special importance. We do not want to waste memory, nor computing power on elements we know to be zero. In comparison to the previously discussed packing algorithms, packing of banded matrices imposes another concern: where the data originates from. This cannot be computed in the same manner as it can in packing algorithms for full matrices. Instead it has to be stored, either on the CPU, or in a separate texture. Krüger and Westermann [KW03] packed banded matrices by packing each diagonal-vector by itself, as shown in Figure 3.3(a). The algorithm is quite simple, and packs sparse banded matrices very efficiently.

Starting at element $(1, 1)$, the algorithm searches for nonzero diagonals in the lower triangular part of the matrix. When a nonzero diagonal-vector

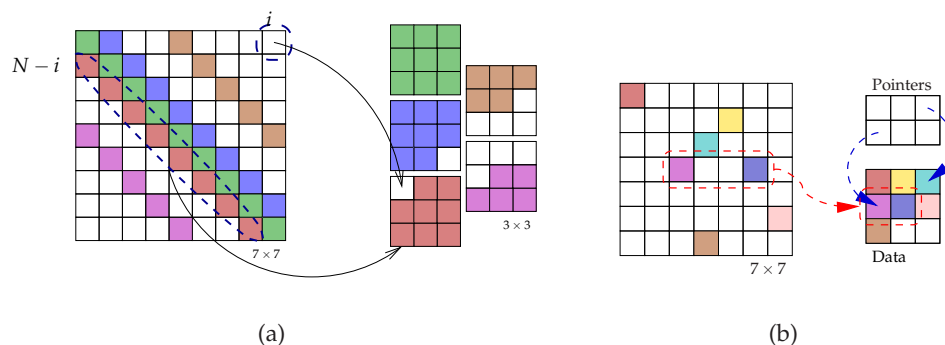


Figure 3.3: Packing of sparse matrices: (a) Packing using diagonal vectors as proposed by Krüger and Westermann [KW03]. (b) Packing of unstructured sparse matrices, presented by Bolz et al. [BFGS03]. The red dotted line shows how all nonzero entries in one row is consecutively stored in a texture. The blue dotted lines show the pointers from the pointer texture to the first nonzero element in each row.

is found, it is joined with the corresponding elements from the upper triangular part of the matrix, creating a vector of length N . When all diagonals have been found, the algorithm terminates. The diagonal is always packed as a separate texture, even though it might theoretically contain only zeros.

Unstructured sparse matrices – Krüger and Westermann [KW03] and Bolz et al. [BFGS03] have presented two different ways of storing unstructured sparse matrices. Krüger and Westermann [KW03] used vertex arrays to represent all nonzero entries of a matrix. Each element in the matrix is rendered as a single vertex, resulting in a single fragment at the correct position. In contrast, Bolz et al. [BFGS03] presented a way of storing unstructured sparse matrices using two textures. The first texture contains all nonzero elements, stacked consecutively row by row (see Figure 3.3(b)). The second texture contains a pointer to the first element from each row in the first texture. However, these two ways of storing matrices have to be recomputed if the nonzero structure changes. Dynamic sparse data-representations on the GPU have also been presented, and I refer to Glift [LSK⁺06], Lefohn et al. [LKHW04] and Strzodka and Telea [ST04] for details.



3.2 GPU optimizations

Because the GPU is a closed architecture [JS05], it is not publicly known exactly how the cache and other important parts of the hardware operates.

Nevertheless, there has been research in this field as well, in order to point out bottlenecks and possible optimizations.

APIs such as Compute Unified Device Architecture (CUDA) [NVI07b] from NVIDIA and Close To the Metal (CTM) [Adv06] from ATI are so close to the underlying hardware that some details can be extrapolated from user guides. Chapters, such as “Performance Guidelines” often give a pointer to how the underlying hardware operates with respect to cache sizes, etc. Because these APIs are not available for the hardware used in this thesis, I refer the reader to the CUDA Programming Guide [NVI07b] and the ATI CTM Guide [Adv06] for further information.

GPU texture pre-fetch – The GPU utilizes texture coordinates to look up textures. It also uses these texture coordinates to pre-fetch data into cache and memory close to the GPU itself. By pre-computing the texture coordinates on the CPU before assigning them to the textures, as opposed to computing them on the fly in the fragment processor, the GPU can pre-fetch the data faster. This optimization can lead to a speed increase of up to 25 percent [GGHM05].

Blocking of matrices – Another optimization, which several authors [HCH03, FSH04, JS05, GLGM06] have pointed to, is blocking of matrices. Most authors agree that blocking increases performance. Blocking on the GPU is the same technique as used on the CPU; by restricting computation to a limited area, thereby preventing cache flushes, the cache hit ratio is expected to rise.

3.3 Numerical linear algebra

Linear algebra is a fundamental part of many algorithms [Kni04], and fits the GPU programming model well. There has been a lot of research using the GPU to solve linear algebra, and the main results presented have been matrix multiplication, PLU factorization and the conjugate gradients algorithm.

3.3.1 Matrix multiplication

Several authors have presented matrix-matrix multiplication on the GPU. Larsen and McAllister [LM01] were the first to present matrix multiplication on the GPU. Using the fixed function pipeline, they computed the product using a multi-pass approach, and reported the same number of operations per second as the ATLAS [WD98b] implementation. However, the hardware was limited to only 8 bits of precision, and they also pointed to bandwidth as the limiting factor.

Hall et al. [HCH03] presented a multichannel matrix multiplication algorithm using the two-by-two packing algorithm mentioned previously. In addition, they used blocking to increase the number of cache hits, and reported 25% less data transfer and lower instruction count compared to the implementation of Larsen and McAllister [LM01]. The blocking technique simply restricted the input of the computation, and used multiple passes instead.

Moravánszky [Mor03] presented a way of multiplying two matrices by packing four-by-one sub-matrices into a single pixel. Because of hardware instruction restraints, the algorithm was decomposed into several consecutive passes, that each computed part of the result.

Fatahalian, Sugerman and Hanrahan [FSH04] also presented dense matrix-matrix multiplication. In their publication, they presented both a single-pass and a multi-pass algorithm. The single pass algorithm used a for-loop executed in a shader to compute the result of each output element. The multi-pass algorithm, on the other hand, was a modification to the algorithm presented by Larsen and McAllister [LM01], where they enhanced it by packing four-by-one sub-matrices into each pixel as described by Moravánszky [Mor03]. They reported that their implementation was bounded by bandwidth, but that it was faster than ATLAS, even when incorporating the time used to transfer and pack data.

Jiang and Snir [JS05] presented an automatic tuning matrix multiplication on the GPU. By benchmarking the effect of different optimizations, they searched through the results for an optimal setup for the specific underlying hardware. The parameters they benchmarked included using multiple render targets, using different packing schemes, varying the number of passes to use, unrolling loops, changing compiler (DirectX [Mic07b] or Cg [NVI07a]) and using different compiler settings. They reported performance comparable to hand tuned versions for multiple hardware setups.

Govindaraju et al. [GLGM06] presented an analysis of the effect of blocking to increase the number of cache hits. They blocked the output size of each computation, and reported less than 6% cache misses when using their blocking algorithm.

3.3.2 PLU factorization

LU factorization is a way of solving a linear system of equations, $Ax = b$. PLU factorization includes pivoting for numerical stability.

Galoppo et al. [GGHM05] presented a way of computing both the LU and PLU factorization of a matrix using the GPU. They stored the matrices in a single component texture, and used the ping-pong technique to sequentially reduce the matrix. Their algorithm for PLU factorization consisted of five parts that were executed in each pass:

1. Find the pivot element by rendering one fragment that loops through the pivot search area.
2. Swap the pivot row with the current top row. This is done by rendering the pivot row from the source buffer to the position of the top row in the destination buffer. The top row is similarly swapped in the same pass, and the two rows are finally copied back to the source buffer.
3. Copy pivot row to destination buffer from source buffer.
4. Normalize the copied row and copy back to the source buffer.
5. Update the remaining lower right part of the matrix and write to the destination buffer.

When one pass has completed, the role of the source and destination buffers were reversed, so that the source buffer became the new destination buffer.

Their algorithm for computing the LU factorization simply removed the two first steps. In their PLU factorization, however, they implemented both partial, and full pivoting. While partial pivoting searches for the pivot element in the column of the pivot position, full pivoting searches for the pivot element throughout the rest of the matrix.

Full pivoting does not only swap rows, but also columns, which is done in a similar fashion as described for the partial pivoting. But finding the index of the pivot element is far more complicated, and is computed using multiple shaders and passes.

Because of the complex pivoting and swapping strategies, the algorithm has to fetch 3 floats per MAD instruction [GGHM05]. In addition, storing the data in one color channel effectively only utilizes part of the full four-way vectorized arithmetic available in most GPUs. Their benchmarks, however, reported the algorithm as faster than the highly optimized ATLAS routines. Nevertheless, the benchmarks were synthetic, and assumed no cache misses.

3.3.3 Conjugate gradients

Conjugate gradients is a non-stationary [Lyc06] iterative matrix solver used to solve a positive definite linear system $Ax = b$. It is also a direct method that can find the exact¹ solution in a finite number of iterations. Nevertheless, it is most often used as an iterative solver as it usually requires few iterations for sufficiently accurate results.

¹Only exact when using exact arithmetic. Floating point rounding errors are inevitable.

Bolz et al. [BFGS03] presented conjugate gradients computed on the GPU. They represented the sparse matrices using two textures; one texture of pointers, and one with the data stacked row by row sequentially. Using this data-structure, they implemented the matrix-vector product and the vector-vector inner product used in the conjugate gradients algorithm. Their performance results, however, showed that their algorithm was heavily penalized by data transfer to and from the CPU, and the random data access on the GPU which flushed the cache constantly.

Krüger and Westermann [KW03] also implemented conjugate gradients. By storing all nonzero diagonal-vectors in separate textures and using their implementation of basic linear algebra operators, they reported precision issues, and a speedup over unoptimized CPU code.

Chapter 4

A Numerical Linear Algebra Interface to the GPU for MATLAB

“To boldly go where no one has gone before”
— Star Trek: The Next Generation

In this chapter, I present my implementation on the GPU of three algorithms from numerical linear algebra. I also present an interface between this implementation and MATLAB, which enables use of the GPU as a coprocessor, computing in the background.

4.1 Algorithms

I present three algorithms: matrix-matrix multiplication, Gauss-Jordan elimination, and PLU factorization. Matrix-matrix multiplication is one of the major building blocks in linear algebra, used in numerous algorithms. Gauss-Jordan elimination and the PLU factorization of a matrix both solve a linear system, $Ax = b$. I have chosen to implement these specific algorithms because of their parallel nature, and their importance in the field of numerical linear algebra.

4.1.1 Matrix-matrix multiplication

Matrix-matrix multiplication is an operation that occurs in many mathematical problems, including matrix solvers and others. It is a computationally demanding process in terms of memory bandwidth and processor ca-

capacity. The product is defined as

$$(AB)_{i,j} = \sum_{k=1}^n a_{i,k} b_{k,j}, \quad A \in \mathbb{R}^{n,m}, \quad B \in \mathbb{R}^{o,n}. \quad (4.1)$$

The standard way of computing the product of two $n \times n$ matrices has a computational and memory access of order $\mathcal{O}(n^3)$. The fastest known algorithm, published by Coppersmith et al. [CW87], is of order $\mathcal{O}(n^{2.38})$, but most researchers agree that $\mathcal{O}(n^2)$ is optimal [Rob05]. However, these accelerated algorithms impose large overheads hidden in the $\mathcal{O}(n)$ notation [Hig02]. Algorithms with an exponent lower than 2.775 have such large hidden overheads that they would only beat the standard $\mathcal{O}(n^3)$ algorithm for gigantic n . Because the algorithms are used recursively, the ones with an exponent higher than, or equal, to 2.775 are also impractical for implementation on the GPU because of the large overhead connected with starting a computation and context switches.

If we however assume that an algorithm of order $\mathcal{O}(n^2)$ is available, and of practical use, the matrix-matrix product will still be an expensive operation for large n . Since the problem is embarrassingly parallel, in the sense that one output element only depends on one column and one row from the input matrices, we can utilize the immense speed of the GPU to compute it.

There are several ways of viewing the matrix product. By viewing the problem from a specific angle, we can possibly discover a structure beneficial for our specialized SIMD hardware. I will now describe two distinct ways of viewing the sum in (4.1).

Vector-vector inner product – Instead of viewing one output element as the sum in (4.1), we can view the sum as the vector-vector inner product of one row from A , and a column from B :

$$(AB)_{i,j} = a_i^T b_j, \quad a_i^T \in \mathbb{R}^n, \quad b_j \in \mathbb{R}^n, \quad (4.2)$$

where a_i^T is the i^{th} row vector of A , and b_j is the j^{th} column vector of B . We can assign a virtual processor to each element in the output matrix, which then can start computation independent of all other processors. However, this algorithm does not parallelize all the work.

A virtual cube of processors – The vector-vector inner product itself can also be computed in parallel. This is done using a virtual cube of processors, which all compute a small part of the result. A schematic of the algorithm is shown in Figure 4.1. This way of computing the matrix-matrix product is far more parallel than the vector-vector inner product, because more of the work is distributed to the processing nodes.

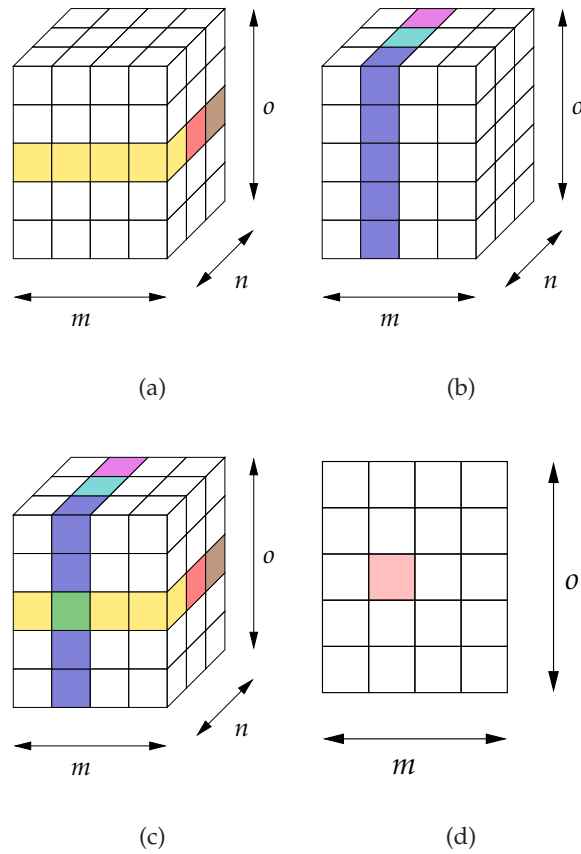


Figure 4.1: The matrix-matrix multiplication algorithms visualized using a virtual cube of processors: (a) A is replicated throughout the cube in the m dimension. (b) B is replicated throughout the cube in the o dimension. (c) Each processor has received a single element from A and B , and computes the result of these elements. (d) The cube is then summed over the n dimension to compute the final result.

Instead of summing up after all nodes have computed their partial result, we can sum as we go along. This reduces the amount of total memory needed by a factor n , and yields an optimally parallel algorithm, where all operations have been separated into independent tasks.

Implementation

I have implemented both the vector-vector inner product, and the virtual cube of processors approach to matrix multiplication. One has more texture IO, while the other clusters more computation to each processor. This enables the analysis of cache efficiency versus processor speed.

Choice of packing algorithm – The data is represented on the GPU in form of textures. As discussed in Section 3.1, it is vital to pack data in an efficient way for the specific application. An efficient way to pack data for this application would be to store A as a four-by-one packed texture, and B as a one-by-four packed texture. This enables us to write the sum as

$$(AB)_{i,j} = \sum_{k=0}^{n/4-1} [a_{4k+1} \ a_{4k+2} \ a_{4k+3} \ a_{4k+4}]_i \begin{bmatrix} b_{4k+1} \\ b_{4k+2} \\ b_{4k+3} \\ b_{4k+4} \end{bmatrix}_j. \quad (4.3)$$

Since the GPU has implemented vectorized operations to execute four MAD instructions in parallel per clock cycle, we can compute each output element in $n/4$ clock cycles (assuming no IO wait or other stalls). The packing is also efficient with respect to the data access pattern for this algorithm, since it increases the number of matrix elements per row and column in cache for A and B respectively.

But even though this packing might be optimal for this specific application, it requires substantial overhead when reusing data from previous computations. If A or B is stored on the GPU using another packing scheme, we have to repack it before we can even start the computation. This overhead is substantial, and should be avoided. To avoid it, I have implemented matrix-matrix multiplication using the two-by-two packing scheme shown in Figure 3.1(c). All other algorithms will also take two-by-two packed textures as input, and result in two-by-two packed textures. Optimally, the toolbox should not be locked to only one packing scheme, but implementing algorithms for all input and output packing schemes is a task beyond this thesis. I have selected to use the two-by-two packing because it is a good all-round scheme that offers high efficiency for most algorithms.

Instead of computing one output element in parallel, as would be the case with the more optimal packing, we now compute the value of small

two-by-two sub-matrices:

$$= \sum_{k=0}^{n/2-1} \begin{bmatrix} a_{i,2k+1} & a_{i,2k+2} \\ a_{i+1,2k+1} & a_{i+1,2k+2} \end{bmatrix} \begin{bmatrix} (AB)_{i,j} & (AB)_{i,j+1} \\ (AB)_{i+1,j} & (AB)_{i+1,j+1} \end{bmatrix} \begin{bmatrix} b_{2k+1,j} & b_{2k+1,j+1} \\ b_{2k+2,j} & b_{2k+2,j+1} \end{bmatrix}. \quad (4.4)$$

This enables us to compute four elements in the output matrix in n clock cycles (again, disregarding IO wait and other stalls).

Vector-vector inner product – The vector-vector inner product implementation uses the two-by-two packing algorithm to store the matrices, as motivated in the previous paragraph. The algorithm completes in a single rendering pass, which is executed as follows:

1. First, texture coordinates are set to cover the entire input matrices, A and B . We can use the same coordinates for both A and B without their dimensions having to match, as shown in Figure 4.2(a).
2. Then, we render a large quad covering the whole of our output matrix, $C = (AB)$. Each of the fragments in the output texture have an interpolated texture-coordinate (see Section 2.2), $\langle s, t \rangle$, which corresponds to the same position in matrices A and B .
3. We utilize the fact that C has the same number of columns and rows as B and A , respectively. Due to this fact, we can find the position of every element in row t in A and column s in B .
4. Using consecutive for-loops (as discussed in Subsection 2.2.5), the fragment shader computes the inner product of row t from A and column s from B .

This algorithm fetches $2n$ two-by-two matrices, and computes $2n$ vectorized multiply and add instructions per output two-by-two matrix. This gives us a texture IO to MAD-instruction ratio of 1:1.

The fragment shader in Listing 4.1 is used to compute the result. It is in the for-loop that we compute the vector-vector inner product of two two-by-two sub-matrices. We start by computing the texture-coordinates to the k^{th} sub-matrix in the row-vector from A and the column-vector from B . Notice that the texture coordinates are computed as

$$k = \frac{k + 0.5}{n}, \quad (4.5)$$

where adding 0.5 places us in the middle of a texel. Then we fetch the two-by-two sub-matrices, and compute the multiplication:

$$\left(\begin{bmatrix} r & g \\ b & a \end{bmatrix}_A \begin{bmatrix} r & g \\ b & a \end{bmatrix}_B \right) = \begin{bmatrix} r_A r_B + g_A b_B & r_A g_B + g_A a_B \\ b_A r_B + a_A b_B & b_A g_B + a_A a_B \end{bmatrix} \quad (4.6)$$

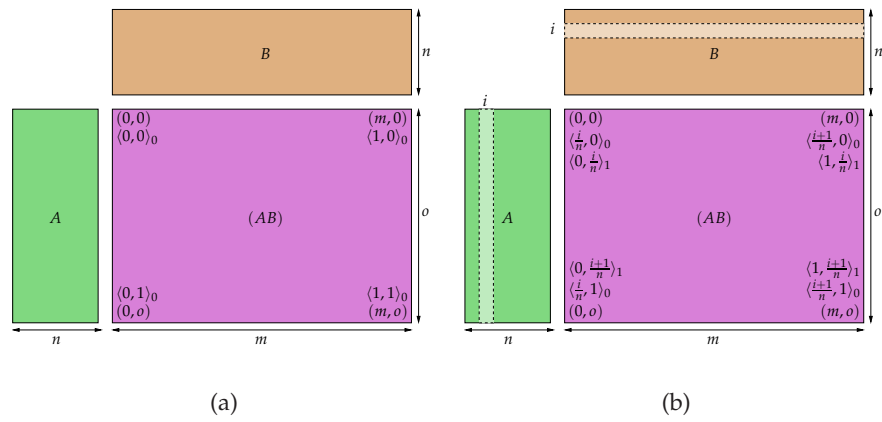


Figure 4.2: Schematic showing input textures, and output texture for matrix multiplication. The purple quad represents the output matrix computed from the input matrices represented by the green and brown quads. $\langle s, t \rangle_l$ denotes that texture coordinate l is given the value $\langle s, t \rangle$. (x, y) denotes the position of a vertex in the quad covering the output matrix: (a) Vertex and texture coordinates for vector-vector inner product algorithm. (b) Coordinates for virtual cube of processors algorithm in pass i of n passes. Here $i = 0, \dots, n - 1$ is the texture coordinate to the intersection between row i and $i - 1$, thus setting the texture coordinates to cover the lightly shaded areas.

Listing 4.1: Vector-vector inner product shader

```

1 uniform sampler2D A; //Texture A
  uniform sampler2D B; //Texture B
  uniform float n;      //Number of two-by-two columns in A
                        //and two-by-two rows in B

5
void main() {
    vec4 C = vec4(0);

    for (float k=0; k<n; ++k) {
10      //Compute the texture coordinate
        //of sub-matrix k in the vectors
        vec2 aCoord = vec2((0.5f + k)/n, gl_TexCoord[0].y);
        vec2 bCoord = vec2(gl_TexCoord[0].x, (0.5f + k)/n);

15      //Fetch sub-matrices
        vec4 aColor = texture2D(A, aCoord);
        vec4 bColor = texture2D(B, bCoord);

        //Compute the product of the two sub-matrices
20      C += aColor.rgbb * bColor.rgrg
          + aColor.ggaa * bColor.baba);
    }

    //If n > 256, consecutive for loops are inserted here,
25 //each of length less than or equal to 256.

    //Set the output color
    gl_FragColor = C;
}

```

This multiplication is computed in shader code using swizzling and smearing, as discussed in Subsection 2.2.4. Finally, the result is stored in the output variable `gl_FragColor`.

Virtual cube of processors – The virtual cube algorithm uses precomputed texture coordinates in a multi-pass shader algorithm, in contrast to the vector-vector inner product algorithm which computes them on-the-fly on the GPU. In each pass of the algorithm, we compute the result of a two-by-two matrix from A and a two-by-two matrix from B for each output fragment. Pass i of the algorithm is shown in Figure 4.2(b), and is executed as follows:

1. In a loop on the CPU, we render a quad covering the whole output matrix, $C = (AB)$, n consecutive times.
2. In the i^{th} iteration, texture-coordinates are set to cover *column* i in A , and *row* i in B . In addition, texture coordinates are set to cover the whole of C . The interpolated texture-coordinates per fragment point to element i in *row* t of A , and element i in *column* s of B , which the GPU can pre-fetch.
3. The fragment processor computes the multiplication of the two two-by-two matrices, and adds their result to the output buffer. Because we are writing to the exact same fragment as we are reading from, we do not need to ping-pong, but must ensure that the quad has been rendered completely before continuing the iteration.

This algorithm has a lot more texture reads and writes than the previously presented approach. Whereas the vector-vector inner product algorithm has $2n$ texture reads, and one texture write per output fragment, this algorithm has $3n$ reads, and n writes. This is a drastic increase, but the precomputed texture-coordinates will decrease the overall texture-wait. Listing 4.2 shows the shader code used to compute the result.

Benchmarking

In order to test the efficiency of the two presented algorithms, I have performed several computations for selected matrix sizes on both the CPU and the GPU.

The ATLAS implementation of the matrix-matrix product in MATLAB has been used to benchmark CPU performance. This implementation is acknowledged as being a highly efficient implementation [MY02]. On the GPU, only the time spent to compute the multiplication is timed, disregarding time used to upload and compile shaders, upload textures, and read

Listing 4.2: Virtual cube of processors shader

```
1 uniform sampler2D A;
  uniform sampler2D B;
  uniform sampler2D acc;

5 void main() {
    //Compute the texture coordinate of elements in
    //A, B and the accumulation buffer
    vec2 aCoord = gl_TexCoord[0].xy;
    vec2 bCoord = gl_TexCoord[1].xy;
10   vec2 cCoord = gl_TexCoord[2].xy;

    //Fetch element i
    vec4 aColor = texture2D(A, aCoord);
    vec4 bColor = texture2D(B, bCoord);
15   vec4 cColor = texture2D(C, cCoord);

    //Compute the product, and add from accumulation
    //buffer
    gl_FragColor = aColor.rrbb * bColor.rgrg
20                  + aColor.ggaa * bColor.baba
                  + cColor.rgba;
}
```

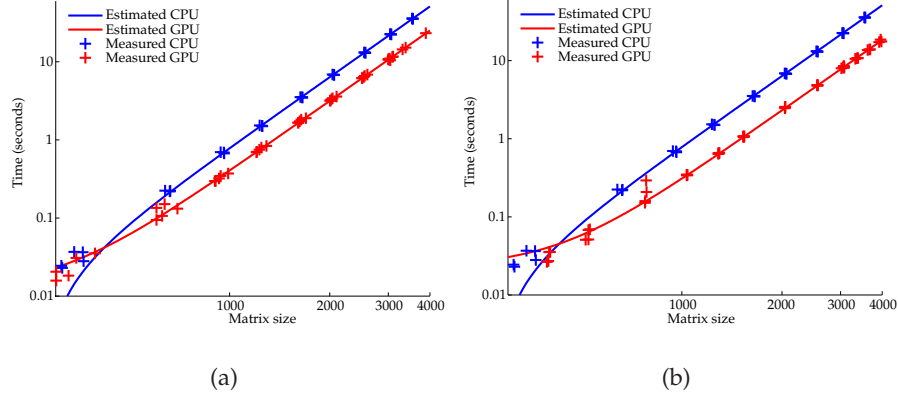


Figure 4.3: Results of benchmarking full matrix-matrix multiplication on the GPU. The crosses are the measured times, while the lines are the functions that fit the sample points best in a least squares sense. The matrix size is given as $\sqrt{\frac{1}{2}(mn + no)}$, where m , n and o have been perturbed by adding a random integer between zero and ten. Notice that both axes are logarithmic, and the size axis starts at 300: (a) The single-pass vector inner product algorithm. (b) The multi-pass virtual cube of processors algorithm.

back textures. This has become a standard way of benchmarking numerical linear algebra on the GPU, used by many authors [GGHM05, KW03, FSH04].

Figure 4.3 shows the measured times to compute the result for both the single- and multi-pass algorithm. In addition, a function that approximates the sampled data-points is also plotted. This function is computed using a least squares approach to the function $a + bx^3$. The exponent is set to three, because the algorithm has the complexity $\mathcal{O}(n^3)$. We do not know whether the MATLAB algorithm is $\mathcal{O}(n^3)$, but the least squares approximation fits the data-points very good over the shown interval. A subset of the measured data-points can be seen in Table B.2.

The computed function to estimate the CPU runtime is

$$-1.74\text{e-}2 + 7.93\text{e-}10 \cdot x^3,$$

and the estimated function for the vector-vector inner product algorithm is

$$1.26\text{e-}2 + 3.93\text{e-}10 \cdot x^3.$$

For large matrices, the coefficient b in $a + bx^3$ will dominate the expression, and we can compute an estimated speedup factor as

$$\text{Speedup} = \frac{b_{\text{CPU}}}{b_{\text{GPU}}}. \quad (4.7)$$

This gives us a speedup factor of 2.02 for the vector-vector inner product algorithm, and is a good estimate for matrices larger than $\sim 700 \times 700$.

If we similarly approximate the measured runtime of the virtual cube of processors algorithm, we end up with the function

$$2.28\text{e-}2 + 2.86\text{e-}10 \cdot x^3.$$

By using the same reasoning, we can estimate the runtime using (4.7), which gives us a factor 2.77. This is a good approximation to the speedup for matrices larger than $\sim 700 \times 700$.

Error analysis

Matrix-matrix multiplication can be stated as $C = (AB)$. To state the error in C as a consequence of a small perturbation of A and B , we start by examining the error in each element [Hig02]. To compute element c_{ij} , we have to compute the inner product of row a_i^T and b_j . This can be stated as

$$\hat{c}_{i,j} = (a_i + \Delta a_i)^T b_j, \quad |\Delta a_i| \leq \gamma_n |a_i|, \quad (4.8)$$

where Δa_i is a small perturbation of row a_i . The backward error for the product is then given as

$$\hat{c}_j = (A + \Delta A_j) b_j, \quad |\Delta A| \leq \gamma_n |A|. \quad (4.9)$$

This states that the computed column j is exact for the perturbed matrix $A + \Delta A_j$. For the full matrix, \hat{C} , the forward error is bounded by

$$|C - \hat{C}| \leq \gamma_n |A| |B|, \quad (4.10)$$

while the backward error is large [Hig02]. This states that our product is sensitive to perturbations in our matrices, and the error is related to the p-norms as

$$\|C - \hat{C}\|_p \leq \gamma_n \|A\|_p \|B\|_p, \quad p = 1, \infty, F. \quad (4.11)$$

When computing with integral matrices, the result has been exact, compared to the double precision ATLAS implementation on the CPU. This can be explained from the error bounds above as there is no error in representing the elements or computing with them. On the CPU, all integers less than or equal to 2^{24} can be represented exactly as IEEE single precision floating point numbers [Wik07a]. The GPU does not implement IEEE single precision, but can represent all numbers smaller than a similarly large number exact. Because of this, we experience no errors in our product for matrices that can be represented, and computed with, exactly on the GPU.

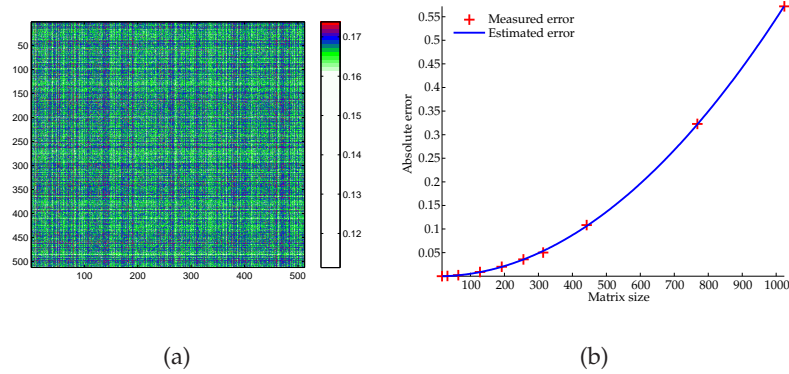


Figure 4.4: Error of full matrix-matrix multiplication on the GPU for uniformly distributed random matrices. The error is computed as $|(AB)_{\text{CPU}} - (AB)_{\text{GPU}}|$: (a) The mean error of five matrix multiplications of two random matrices, uniformly distributed in the interval zero to ten. (b) Error of full matrix-matrix multiplication on the GPU for random matrices as a function of matrix size. The approximated error function is the best fitting second order polynomial in a least squares sense.

For ill-conditioned floating point matrices, however, the error can become larger. The average error of five different matrix multiplications of full random matrices is shown in Figure 4.4. The elements were uniformly distributed between zero and ten. The maximum mean error in any of the computations was 0.14, and the largest error in any of the computations was 0.17.

When plotting the mean absolute error as a function of the matrix size, we see that it closely follows a second order polynomial curve. The relative error, however, is negligible for all matrix sizes ($< 0.001\%$).

4.1.2 Gauss-Jordan elimination

Gauss-Jordan elimination is a standard algorithm in numerical linear algebra, which computes the reduced row echelon form (RREF) of a matrix. MATLAB can compute the RREF of a matrix with the function `rref`. In elementary linear algebra courses where MATLAB is used, the `rref` function is often taught to students as a standard solver for linear systems.

Definition 4.1:

A matrix in reduced row echelon form fulfils the following requirements:

1. Any row consisting of only zeros is located at the bottom.

2. The first coefficient in a non-zero row is always 1.
3. The first coefficient in a non-zero row is always to the right of all leading coefficients in previous rows.
4. All leading coefficients are the only non-zero entries in that column.

The matrix in (4.12), satisfies Definition 4.1, and is on reduced row echelon form:

$$\begin{bmatrix} 1 & 0 & * & 0 & 0 \\ 0 & 1 & * & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad (4.12)$$

Given a linear system $Ax = b$, we can find x by computing the reduced row echelon form of the matrix $C = [A|b]$. Assuming that A is square and invertible, we know that the system $Ax = b$ has a unique solution $\forall b \neq 0$. Since A is invertible we can find its inverse, and

$$Ax = b \quad (4.13)$$

$$A^{-1}Ax = A^{-1}b \quad (4.14)$$

$$x = A^{-1}b \quad (4.15)$$

follows. Because A is invertible, we know that it also has linearly independent rows, which implies that the RREF of A must be the identity matrix. Computing the RREF of C then gives $[I|r]$, where $r = x = A^{-1}b$.

Gauss-Jordan elimination – Gauss-Jordan elimination is a simple modification of Gaussian elimination. It is slightly more computationally complex compared to Gaussian elimination, but fits the GPU programming model better because of less buffer switches.

Starting with a linear system $Ax = b$, where $A \in \mathbb{R}^{5,5}$ is invertible and $b \in \mathbb{R}^{1,5}$, we compose our matrix C as

$$C = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & b_1 \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & b_2 \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & b_3 \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & b_4 \\ a_{5,1} & a_{5,2} & a_{5,3} & a_{5,4} & a_{5,5} & b_5 \end{bmatrix}. \quad (4.16)$$

To find our x using Gauss-Jordan elimination, we start by normalizing the first row with respect to $a_{1,1}$. Then we eliminate all other entries in the first

column by subtracting multiples of the first row. This gives us

$$\begin{bmatrix} 1 & i_{1,2} & i_{1,3} & i_{1,4} & i_{1,5} & i_{1,6} \\ 0 & i_{2,2} & i_{2,3} & i_{2,4} & i_{2,5} & i_{2,6} \\ 0 & i_{3,2} & i_{3,3} & i_{3,4} & i_{3,5} & i_{3,6} \\ 0 & i_{4,2} & i_{4,3} & i_{4,4} & i_{4,5} & i_{4,6} \\ 0 & i_{5,2} & i_{5,3} & i_{5,4} & i_{5,5} & i_{5,6} \end{bmatrix}, \quad (4.17)$$

where $i_{i,j}$ are intermediate results. We continue by normalizing the second row with respect to $i_{2,2}$, and eliminate all other elements in the second column:

$$\begin{bmatrix} 1 & 0 & i_{1,3} & i_{1,4} & i_{1,5} & i_{1,6} \\ 0 & 1 & i_{2,3} & i_{2,4} & i_{2,5} & i_{2,6} \\ 0 & 0 & i_{3,3} & i_{3,4} & i_{3,5} & i_{3,6} \\ 0 & 0 & i_{4,3} & i_{4,4} & i_{4,5} & i_{4,6} \\ 0 & 0 & i_{5,3} & i_{5,4} & i_{5,5} & i_{5,6} \end{bmatrix}. \quad (4.18)$$

We continue this process, normalizing and eliminating until we are left with

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & r_1 \\ 0 & 1 & 0 & 0 & 0 & r_2 \\ 0 & 0 & 1 & 0 & 0 & r_3 \\ 0 & 0 & 0 & 1 & 0 & r_4 \\ 0 & 0 & 0 & 0 & 1 & r_5 \end{bmatrix}, \quad (4.19)$$

where $x_i = r_i$ is the solution to our system.

Numerical stability – This algorithm is numerically unstable if $a_{1,1}$ is sufficiently close to zero in the first step, or $i_{k,k}$ is sufficiently close to zero in step k . If the element is zero, the algorithm will fail completely. To increase the stability of the algorithm, a strategy known as pivoting is employed. Pivoting interchanges rows and/or columns; interchanging rows is known as partial pivoting, while interchanging rows and columns is known as complete pivoting. The following example, loosely transcribed from [Mey04], shows how pivoting increases numerical stability. We start with the linear system

$$\begin{bmatrix} -10^{-4} & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \quad (4.20)$$

Eliminating element (2,1) gives us

$$\begin{bmatrix} 1 & -10^4 \\ 0 & 1 + 10^4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} -10^4 \\ 2 + 10^4 \end{bmatrix}. \quad (4.21)$$

If we now try to solve this system using three significant digits we will end up with

$$x_2 = \frac{2 + 10^4}{1 + 10^4} \stackrel{\text{float}}{=} \frac{10000}{10000} = 1 \quad (4.22)$$

$$x_1 = -10^4 + 10^4 x_2 \stackrel{\text{float}}{=} -10000 + 10000 = 0. \quad (4.23)$$

If we, however, swap the two rows so that the system reads

$$\begin{bmatrix} 1 & 1 \\ -10^{-4} & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \quad (4.24)$$

and eliminate element (2,1), we end up with

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 + 10^{-4} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 2 \\ 1 + 2 \cdot 10^{-4} \end{bmatrix}, \quad (4.25)$$

which has the solution

$$x_2 = \frac{1 + 2 \cdot 10^{-4}}{1 + 10^{-4}} \stackrel{\text{float}}{=} \frac{10000}{10000} = 1 \quad (4.26)$$

$$x_1 = 2 - x_2 \stackrel{\text{float}}{=} 2 - 1 = 1. \quad (4.27)$$

This is the exact solution to three digits, in contrast to the un-pivoted solution which was far off.

The following matrix shows terminology connected with pivoting. The red shaded area is referred to as the pivot row, where the first element is called the pivot position. The green shaded area is referred to as the pivot search area:

$$\begin{bmatrix} 1 & 0 & i_{1,3} & i_{1,4} & i_{1,5} & i_{1,6} \\ 0 & 1 & i_{2,3} & i_{2,4} & i_{2,5} & i_{2,6} \\ 0 & 0 & p_{3,3} & i_{3,4} & i_{3,5} & i_{3,6} \\ 0 & 0 & i_{4,3} & i_{4,4} & i_{4,5} & i_{4,6} \\ 0 & 0 & i_{5,3} & i_{5,4} & i_{5,5} & i_{5,6} \end{bmatrix} \quad (4.28)$$

In complete pivoting, we search for the maximum element within the dotted box, i.e., the pivot row and the pivot search area. Then the column of the pivot position, and the column containing the maximum are interchanged. The rows are also interchanged, in order to bring the maximum element to the pivot position.

In partial pivoting, we search for the maximum within the dashed box, i.e., we find the maximum of the pivot element and the first column of the pivot search area. Then, the row containing the maximum and the pivot row are interchanged.

Implementation

The implementation of the RREF algorithm uses OpenGL to trick the GPU into thinking it is only working on graphical primitives. The algorithm is implemented as a multi-pass shader, where the passes are computed sequentially in a ping-pong fashion. To create a numerically stable algorithm, a novel algorithm for partial pivoting is used.

Pivoting – In general, we want to swap the pivot row with the row containing the largest element. But since we have packed our data into two-by-two sub-matrices, we swap rows of two-by-two sub-matrices. For a general two-by-two matrix,

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \quad (4.29)$$

to be considered as a candidate for pivoting, we require that $a \neq 0$. In addition, we have to require that $d \neq bc/a$, because this would lead to $A_{2,2}$ being zero after eliminating c .

To find the best suited two-by-two row, we employ a reduction shader that consists of two steps. In the first step, we read from buffer `a`, and write to the `pivotA` buffer. In this step we calculate a measure of suitedness for each two-by-two sub-matrix. This is done by calculating a norm of the diagonal-elements after elimination,

$$A' = \begin{bmatrix} a & 0 \\ 0 & d - bc/a \end{bmatrix}. \quad (4.30)$$

I present two different norms that have slightly different properties. The two norms,

```
geom = a11*a22;
```

and

```
harm = a11*a22/(a11+a22);
```

where

```
a11 = abs(a);
```

```
a22 = abs(d - b*c/a);
```

are plotted in Figure 4.5. The first norm, `geom` is related to the geometric mean of the two numbers, while `harm` is related to the harmonic mean of the two numbers. Only one of the two norms are used, but I mention them both because of their numerical properties. As shown in the figure, `geom` has a relatively low resolution in the lower values, and a high resolution in high values. The opposite is true for `harm`. If we only have large values, it is of less importance which of the large values we choose, but for only

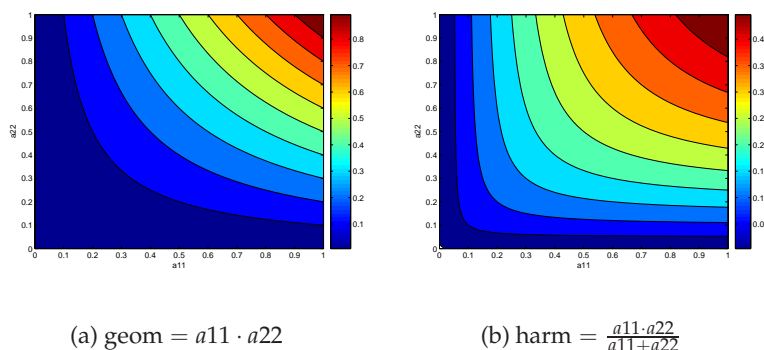


Figure 4.5: The two norms, *geom* and *harm*, plotted in the same coordinate system. Notice that while *geom* has high resolution in the large values, *harm* has high resolution in the low values.

low values, it is vital that we choose the largest possible value for optimal numerical stability. Because we are working in single precision only, we should use *harm*, even though *geom* might be satisfactory for most matrices.

In addition to simply computing the norm, the shader also reduces the data by four. Four texels, each representing a two-by-two matrix, are fetched, and the norm is computed for each. The largest of the four norms, with the accompanying row index is written to the output buffer. Finding the pair, however, is not trivial on the GPU. Since the GPU operates on elements in vectors, and not vectors as a whole, we cannot find the maximum *and* the coordinate in one operation. Instead, we first find the maximum, and then locate the coordinate. The maximum is found using the following shader code:

```
maximum = max(norm1, max(norm2, max(norm3, norm4)));
```

Finding the coordinate corresponding to the maximum norm is the hardest part of the search algorithm. The naïve approach,

```
if (norm1 == maximum) { coord = coord1; }
else if (norm2 == maximum) { coord = coord2; }
else if (norm3 == maximum) { coord = coord3; }
else { coord = coord4; }
```

is a major bottleneck, because many processors will have to execute branches they do not take, as mentioned in Subsection 2.2.5. In stead of using the explicit if-tests, we can rewrite the shader to use implicit if-tests:

```

coord = max(float(norm1 == maximum)*coord1,
            max(float(norm2 == maximum)*coord2,
                max(float(norm3 == maximum)*coord3,
                    float(norm4 == maximum)*coord4
                )
            )
);

```

Here, each `float(x == y)` statement is an implicit if-test. It returns a boolean, which is then cast to a float, yielding 0.0f or 1.0f. This implicit test is then multiplied with the appropriate coordinate, giving the two possible outcomes, 0 or `coordi`. Since we know `maximum` is equal to at least one of the four norms, we are guaranteed to get the maximum index of the largest norm.

The second step simply searches through the rest of our search domain, and ends up with the largest norm with the corresponding row index. This is done using a reduction operator that operates in the same manner as the reduction in the first step.

Because it is possible that we have no elements that fulfill our requirements, we require that our norm is larger than ϵ at the end of the computation. A norm which is less than ϵ , corresponds to an invalid pivot element. Here, ϵ should be chosen to be close to the machine epsilon of the GPU. Thus if the final norm is less than ϵ , we assume that the row contains only zeros, and thus is already reduced. This also implies that the matrix is singular, or near singular.

Algorithm – The GPU algorithm is a multi-pass algorithm, depicted in Figure 4.6. First, the maximum element in the pivoting column is located in $\log(n - i) / \log(4)$ passes (see Subsection 2.2.4) as described in the previous paragraph. Then the row containing the maximum and the pivot row are swapped, as shown in Figure 4.6(a). Finally, all elements above and below the pivot position are eliminated, shown in Figure 4.6(b).

The pass that swaps the pivot row with the row containing the maximum uses two shaders: `copy` and `normalize`. The `copy` shader is used to copy the green areas in Figure 4.6(a), from the back buffer of the ping-pong buffers, to the front buffer. It is also used to copy from the pivot row to the row containing the maximum.

The pivot row itself is computed using the `normalize` shader. It takes the row containing the maximum value as input, and normalizes it with respect to the first element. Because we are reducing two-by-two matrices, we have to reduce it to reduced row echelon form as well. The RREF of the two rows is computed explicitly from the input,

$$\left[\begin{array}{cc|cc|ccc} r_i & g_i & r_{i+1} & g_{i+1} & \cdots & r_n & g_n \\ b_i & a_i & b_{i+1} & a_{i+1} & \cdots & b_n & a_n \end{array} \right], \quad (4.31)$$

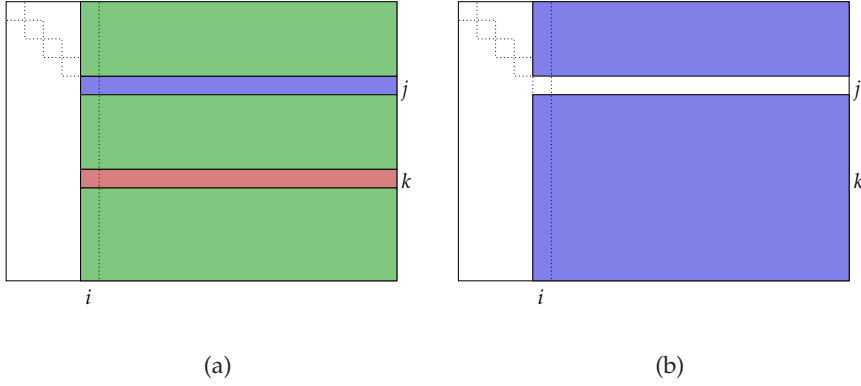


Figure 4.6: The two passes used successively to reduce two columns and two rows of a matrix (the maximum element used for pivoting is found using a reduction shader): **(a)** The first pass normalizes the pivot row, j (blue), and swaps it with the row containing the maximum, k (red). The green area is copied from the last step in the algorithm. **(b)** The second pass eliminates above and below (blue) the pivot row.

by eliminating b_i and g_i , and then normalizing the two rows with respect to r_i and the reduced a_i respectively:

$$\left[\begin{array}{cc|cc|c} 1 & 0 & \alpha r_{i+1} - \delta \gamma (b_{i+1} - \beta r_{i+1}) & \alpha g_{i+1} - \delta \gamma (a_{i+1} - \beta g_{i+1}) & \cdots \\ 0 & 1 & \gamma (b_{i+1} - \beta r_{i+1}) & \gamma (a_{i+1} - \beta g_{i+1}) & \cdots \\ \cdots & & \cdots & \cdots & \\ \cdots & & \alpha r_n - \delta \gamma (b_n - \beta r_n) & \alpha g_n - \delta \gamma (a_n - \beta g_n) & \end{array} \right]. \quad (4.32)$$

The two rows in 4.32 fulfil the RREF definition (Definition 4.1), where the coefficients α, β, γ , and δ are computed as

$$\begin{aligned} \alpha &= \frac{1}{r_i} & \beta &= \frac{b_i}{r_i} \\ \gamma &= \frac{1}{a_i - \frac{b_i g_i}{r_i}} & \delta &= \frac{g_i}{r_i}. \end{aligned} \quad (4.33)$$

The vertex shader is used to compute the coefficients α, β, γ , and δ as shown in Listing 4.3. They are transferred to the fragment shader using the varying variable `normConstant`. Because it is evaluated equally in all four vertices in the quad, the interpolated value will also be the same for all produced fragments. The fragment processor then uses these values to normalize the whole row as shown in Listing 4.4.

In the second pass, we use only one shader, `eliminate`. Remember that the front and back buffers have been swapped, so the back buffer now

Listing 4.3: Vertex shader that computes normalizing constants.

```

1 uniform sampler2D INPUT;      //Input row
  varying vec4 normConstant;    //Normalizing constants

  void main() {
5   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0; //The complete row
    gl_TexCoord[1] = gl_MultiTexCoord1; //First texel

    //Look up normalizing texel
10   vec4 normTexel = texture2D(INPUT, gl_TexCoord[1].xy);

    normConstant.r = 1.0f/normTexel.r;          //Alpha
    normConstant.b = normTexel.b / normTexel.r; //Beta
    normConstant.g = 1.0f/(normTexel.a          //Gamma
15      - (normTexel.b*normTexel.g)/normTexel.r);
    normConstant.a = normTexel.g / normTexel.r; //Delta
  }

```

Listing 4.4: Fragment shader which reduces a row of two-by-two matrices to reduced row echelon form.

```

1 uniform sampler2D INPUT;
  varying vec4 normConstant;

  void main() {
5   vec4 thisColor = texture2D(INPUT, gl_TexCoord[0].xy);
    vec4 result;

    //Normalize top row.
    result.rg = normConstant.rr * thisColor.rg;
10

    //Eliminate and normalize bottom row.
    result.ba = normConstant.gg
      * (thisColor.ba - normConstant.bb*thisColor.rg);

15   //Eliminate in top row.
    result.rg = result.rg - normConstant.aa * result.ba;

    //Write result to fragment.
    gl_FragColor = result;
20 }

```

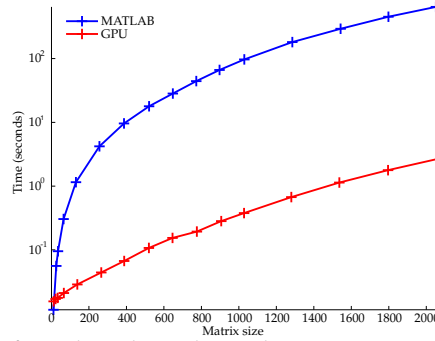


Figure 4.7: Results from benchmarking the GPU RREF algorithm against the implementation used in MATLAB. Notice that the time axis is logarithmic.

contains the result from our previous pass. The `eliminate` shader eliminates the two columns below the pivot position by subtracting a multiple of the pivot row. The multiple we are subtracting with is also computed in the vertex shader, and transferred to the fragment shader as a varying.

Benchmarking

Comparing the internal implementation of RREF used in MATLAB with the presented GPU version showed that the MATLAB implementation must be sub-optimal. The results of this benchmark are shown in Figure 4.7, using a logarithmic time-scale. Users insisting on computing the RREF of a matrix in MATLAB will benefit greatly from using the presented GPU implementation. However, since the implementation of RREF must be sub-optimal, a comparison between the GPU implementation and the highly optimized LU factorization in MATLAB is used instead.

Because LU factorization is a less computationally demanding algorithm than Gauss-Jordan elimination, we still cannot compute an accurate speedup factor. In spite of this, I have compared the two.

Using the same approach as in Section 4.1.1, we can compute a speedup factor of RREF over LU. We start by computing the functions that estimate the runtime best in a least squares sense. The function for the CPU is computed as

$$1.35e-2 + 3.53e-10 \cdot x^3,$$

and the estimated function for the GPU runtime is

$$6.29e-2 + 2.61e-10 \cdot x^3$$

Using (4.7), we can compute the speedup to be 1.35 for large matrices. This speedup factor is a valid approximation for matrices larger than ~ 1200 . A subset of the sampled data-points can be seen in Table B.1.

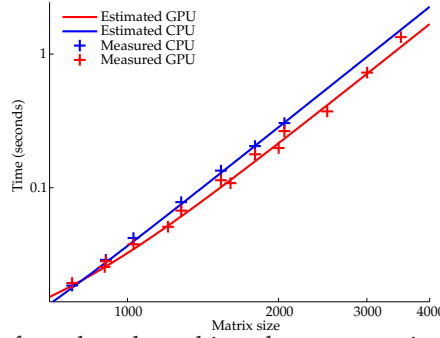


Figure 4.8: Results from benchmarking the RREF against the ATLAS implementation of LU factorization used in MATLAB. Note that LU factorization is a far less computationally and memory heavy operation than RREF. The matrix size is given as $n \times n + 1$. Both axes are logarithmic, and the x-axis starts at 700.

Error analysis

The error analysis for Gaussian elimination is equivalent for all variants, and I refer to Higham for a detailed numerical analysis [Hig02]. To analyze the errors imposed by the single precision GPU algorithm, I solve a system where the solution is known a priori. The test matrix A is uniformly distributed between zero and ten, and the right hand side b is the sum of each row in A . The solution to this system is $x_i = 1, i = 1, \dots, n$.

Figure 4.9 shows the mean error as a function of the matrix size, and shows a distinct linear correlation. The relative error, however, is negligible for all matrix sizes ($< 0.001\%$).

It should be noted that we need to compute the division of two numbers several times in this algorithm. In current graphics hardware, the division a/b is computed by first calculating the reciprocal of b , and then multiplying it with a . This results in two roundoff errors, which makes division an expensive operation when considering the error.

4.1.3 PLU factorization

PLU factorization is an efficient way to solve a system of equations numerically for many right hand sides. It is an LU factorization with a permutation matrix P . The LU factorization is defined as:

Definition 4.2:

The matrices L and U are called the LU factorization of a non-singular matrix A if $LU = A$, L is lower triangular, and U is upper triangular.

We also define the permutation matrix P :

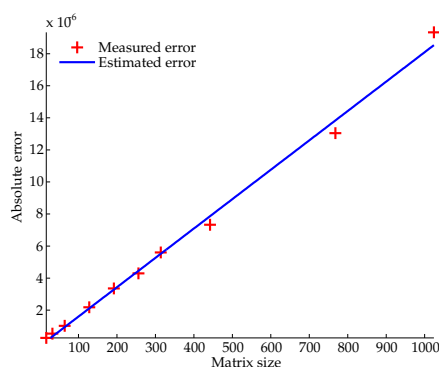


Figure 4.9: Absolute error of b , where $[Ab]$, $A \in \mathbb{R}^{n,n}$, $b \in \mathbb{R}^{n,1}$, as a function of matrix size. The matrix A is uniformly distributed in the interval zero to ten, and b is the row sum of A .

Definition 4.3:

A permutation matrix, P , is the identity matrix with permuted columns.

The PLU factorization is then defined as:

Definition 4.4:

The matrices P , L and U are called a PLU factorization of A if $PLU = A$, P is a permutation matrix, L is a lower triangular matrix, and U is an upper triangular matrix.

The advantage of using a PLU factorization is that the same factorization can be reused for multiple right-hand sides. The system $Ax = b$ is solved as follows, given the PLU factorization of A :

$$\begin{aligned} Pz &= b \\ Ly &= z \\ Ux &= y. \end{aligned} \tag{4.34}$$

First, we permute the columns of b to find z . Then we solve the lower triangular system $Ly = z$ using forward substitution, and then the upper triangular system $Ux = y$ using backward substitution. The forward and backward substitutions both have computational complexity of $\mathcal{O}(n^2)$. For large n , this is a huge difference from the Gauss-Jordan and PLU factorizations, which both are $\mathcal{O}(n^3)$ algorithms.

The Doolittle algorithm

The Doolittle algorithm computes the PLU factorization of a matrix so that L is unit lower triangular ($l_{i,i} = 1, i = 1, \dots, n$), U is upper triangular, and P is a permutation matrix.

The algorithm is a small alteration of Gaussian elimination. In Gaussian elimination, we first reduce our matrix $C = [A|b]$ to an upper triangular matrix using successive forward substitutions:

$$\left[\begin{array}{ccccc|c} * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \\ * & * & * & * & * & * \end{array} \right] \xrightarrow{\text{Forward subst.}} \left[\begin{array}{ccccc|c} * & * & * & * & * & * \\ 0 & * & * & * & * & * \\ 0 & 0 & * & * & * & * \\ 0 & 0 & 0 & * & * & * \\ 0 & 0 & 0 & 0 & * & * \end{array} \right] \quad (4.35)$$

Then, backward substitution is used to compute the solution to the linear system $Ax = b$.

In the PLU factorization we perform forward substitution as well, but on A instead of C . In addition, we save our multipliers, and the permutation resulting from pivoting (see Section 4.1.2). The multipliers will become our L matrix, the permutations become our P matrix, and the upper triangular matrix resulting from the forward substitutions will become our U matrix.

In mathematical terms, we can express the factorization in terms of matrix multiplications [Lyc06]:

$$A_{k+1} = M_k^k P_k A_k, \quad (4.36)$$

where P_k is a permutation matrix, swapping the pivot position with the maximum element in the pivot search area, and

$$\begin{aligned} M_k^k &= P_{k+1} \cdots P_{n-1} M_k P_{n-1} \cdots P_{k+1}, \\ M_k &= I - l_k e_k^T \\ &= I - \begin{bmatrix} 0 \\ r_k \end{bmatrix} e_k^T \\ &= \begin{bmatrix} 1 & \cdots & 0 & \cdots & \cdots & 0 \\ \vdots & \ddots & \vdots & & & \vdots \\ 0 & \cdots & 1 & \cdots & \cdots & 0 \\ \vdots & & -r_1 & \ddots & & \vdots \\ \vdots & & \vdots & & \ddots & \vdots \\ 0 & \cdots & -r_k & \cdots & \cdots & 1 \end{bmatrix}. \end{aligned} \quad (4.37)$$

Here r_i is the multiplier that reduces row $k+i$ in A . By applying this algorithm to our matrix successively, we end up with an upper triangular matrix U :

$$\begin{aligned} U &= M_{n-1}^{n-1} P_{n-1} M_{n-2}^{n-2} P_{n-2} \cdots A \\ &= M_{n-1} P_{n-1} (P_{n-1} M_{n-2} P_{n-1}) P_{n-2} \cdots A \\ &= M_{n-1} \cdots M_1 P_{n-1} \cdots P_1 A_{n-2}, \end{aligned} \quad (4.38)$$

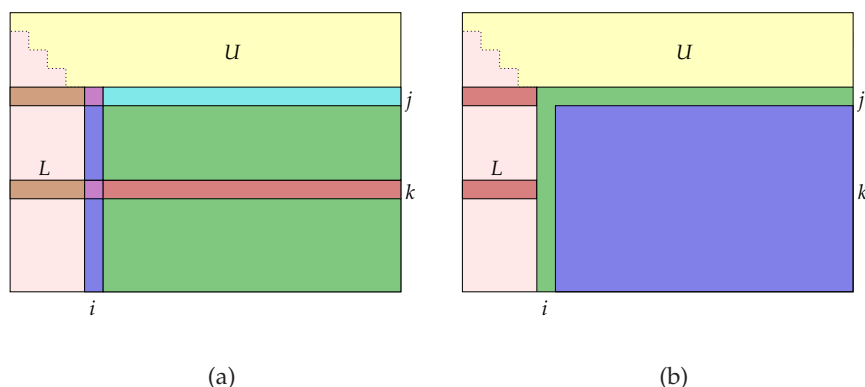


Figure 4.10: The PLU factorization process on the GPU: (a) First pass, computing the multipliers in column i (blue and purple), and swapping the pivot row, j (cyan) with the row containing the maximum, k (red). The previous multipliers (brown) are also swapped to ensure proper structure. (b) Second pass, reducing the rest of the matrix (blue) according to the multipliers computed in the previous pass. The previously computed multipliers are also copied, as well as the recently computed multipliers and top rows (green).

because $P_i P_i = I$. Examining the product $M_{n-1} \cdots M_1$ we see that it is lower triangular, because the product of lower triangular matrices is lower triangular. The inverse of a lower triangular matrix is also lower triangular, and we can write the system as

$$A = P_1 \cdots P_{n-1} M_1^{-1} \cdots M_{n-1}^{-1} U \quad (4.39)$$

$$= PLU \quad (4.40)$$

The inverse, M_k^{-1} , of M_k is explicitly given by changing the sign of r_i . This gives us a very neat structure of L ; it turns out to consist of the multipliers which eliminate the element in A with the same index.

Implementation

The implementation of the Doolittle algorithm on the GPU is executed in much the same way as the RREF algorithm. First, the pivot element is located in a ping-pong fashion as described in Subsection 4.1.2. Then, we compute the multipliers, and finally reduce the rest of the matrix according to the multipliers, shown in Figure 4.10.

Memory optimization – We know that L is unit lower triangular, thus the diagonal entries are all 1. By exploiting this, we can save memory by stor-

ing both matrices in one texture:

$$\begin{aligned}
 & \begin{bmatrix} l_{1,1} & 0 & 0 & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 & 0 & 0 \\ l_{3,1} & l_{3,2} & l_{3,3} & 0 & 0 \\ l_{4,1} & l_{4,2} & l_{4,3} & l_{4,4} & 0 \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & l_{5,5} \end{bmatrix} + \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} \\ 0 & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} \\ 0 & 0 & u_{3,3} & u_{3,4} & u_{3,5} \\ 0 & 0 & 0 & u_{4,4} & u_{4,5} \\ 0 & 0 & 0 & 0 & u_{5,5} \end{bmatrix} \\
 &= \begin{bmatrix} l_{1,1} & 0 & 0 & 0 & 0 \\ 0 & l_{2,2} & 0 & 0 & 0 \\ 0 & 0 & l_{3,3} & 0 & 0 \\ 0 & 0 & 0 & l_{4,4} & 0 \\ 0 & 0 & 0 & 0 & l_{5,5} \end{bmatrix} + \begin{bmatrix} u_{1,1} & u_{1,2} & u_{1,3} & u_{1,4} & u_{1,5} \\ l_{2,1} & u_{2,2} & u_{2,3} & u_{2,4} & u_{2,5} \\ l_{3,1} & l_{3,2} & u_{3,3} & u_{3,4} & u_{3,5} \\ l_{4,1} & l_{4,2} & l_{4,3} & u_{4,4} & u_{4,5} \\ l_{5,1} & l_{5,2} & l_{5,3} & l_{5,4} & u_{5,5} \end{bmatrix} \quad (4.41)
 \end{aligned}$$

This more than halves the memory needed to represent the data, and it is trivial to reconstruct the two matrices from the packed representation.

Multipliers – In the first pass after locating the pivot element, we swap the pivot row j , with the row k containing the maximum. Again, since we are using two-by-two packing, we have to swap two two-by-two rows. In addition, we have to compute the multiplier of the second row of the first two-by-two element, and eliminate element $(2, 1)$:

$$\left[\begin{array}{cc|c} r_{j,i} & g_{j,i} & \cdots \\ b_{j,i} & a_{j,i} & \cdots \end{array} \right] \xrightarrow{\text{Elimination}} \left[\begin{array}{cc|c} r_{j,i} & g_{j,i} & \cdots \\ 0 & a_{j,i} - l g_{j,i} & \cdots \end{array} \right]. \quad (4.42)$$

The multiplier $l = r_{j,i}/b_{j,i}$ is computed by the vertex shader shown in Listing 4.5. The computed α is then used to reduce the rest of the row. This shader is used to compute row j (cyan) in Figure 4.10(a).

But because of our memory representation, mentioned in the previous paragraph, we need to store the multiplier, l , in position $(2, 1)$ of the pivot position two-by-two matrix, element (j, i) (purple) in Figure 4.10(a).

To compute the multipliers in the rest of column i , we use a vertex shader that computes the pivot element, shown on the right hand side of (4.42). Using these values, we can compute the multipliers in the column of two-by-two matrices:

$$\begin{aligned}
 & \left[\begin{array}{cc} r_{j+1,i} & g_{j+1,i} \\ b_{j+1,i} & a_{j+1,i} \\ \vdots & \vdots \\ r_{n,i} & g_{n,i} \\ b_{n,i} & a_{n,i} \end{array} \right] \xrightarrow{\text{Elimination}} \left[\begin{array}{cc} l_{j+1}^r & g_{j+1,i} - l_{j+1}^r g_{j,i} \\ l_{j+1}^b & a_{j+1,i} - l_{j+1}^b g_{j,i} \\ \vdots & \vdots \\ l_n^r & g_{n,i} - l_n^r g_{j,i} \\ l_n^b & a_{n,i} - l_n^b g_{j,i} \end{array} \right] \xrightarrow{\text{Elimination}} \left[\begin{array}{cc} l_{j+1}^r & l_{j+1}^s \\ l_{j+1}^b & l_{j+1}^a \\ \vdots & \vdots \\ l_n^r & l_n^s \\ l_n^b & l_n^a \end{array} \right]. \quad (4.43)
 \end{aligned}$$

This is computed using the fragment shader shown in Listing 4.6.

Listing 4.5: Vertex shader computing the multiplier of the top row of two-by-two matrices in the PLU factorization.

```

1 uniform sampler2D INPUT;
  varying float l;

  void main() {
5   gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
    gl_TexCoord[0] = gl_MultiTexCoord0; //The whole row
    gl_TexCoord[1] = gl_MultiTexCoord1; //First element

    //Look up normalizing texel
10   vec4 normTexel = texture2D(INPUT, gl_TexCoord[1].xy);

    //Compute multiplier
    l = normTexel.b / normTexel.r;
  }

```

Listing 4.6: Fragment shader computing the multipliers of a column of two-by-two matrices in the PLU factorization.

```

1 uniform sampler2D INPUT;
  varying vec4 pivot; //Pivot element

  void main() {
5   vec4 thisColor = texture2D(INPUT, gl_TexCoord[0].xy);
    vec4 result;

    //Find multipliers to eliminate the rb column
    result.rb = thisColor.rb / pivot.rr;
10

    //Reduce the ga column
    result.ga = thisColor.ga - result.rb * pivot.gg;

    //Find multipliers to eliminate ga column
15   result.ga = result.ga / pivot.aa;

    //Write result to fragment
    gl_FragColor = result;
  }

```

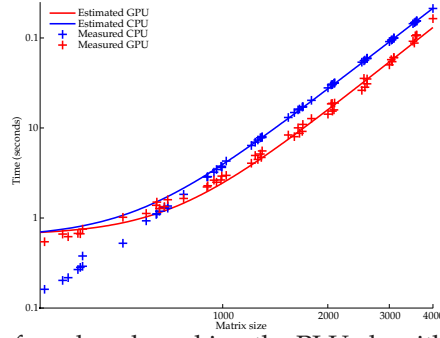


Figure 4.11: Results from benchmarking the PLU algorithm on the GPU compared to the highly optimized ATLAS implementation used in MATLAB. Notice the logarithmic axes. The crosses are the measured times, while the lines are the functions that fit the sample points best in a least squares sense.

Reduction – After computing the multipliers, we reduce the rest of the matrix in Figure 4.10(b) (blue square). Row k , for example, will be computed as

$$\begin{bmatrix} r_{k,i+1} - l_k^r r_{j,i+1} - l_k^s b_{j,i+1} & g_{k,i+1} - l_k^r g_{j,i+1} - l_k^s a_{j,i+1} & \cdots \\ b_{k,i+1} - l_k^b r_{j,i+1} - l_k^a b_{j,i+1} & a_{k,i+1} - l_k^b g_{j,i+1} - l_k^a a_{j,i+1} & \cdots \\ \cdots & r_{k,n} - l_k^r g_{j,n} - l_k^s b_{j,n} & g_{k,n} - l_k^r g_{j,n} - l_k^s a_{j,n} \\ \cdots & b_{k,n} - l_k^b g_{j,n} - l_k^a b_{j,n} & a_{k,n} - l_k^b g_{j,n} - l_k^a a_{j,n} \end{bmatrix}, \quad (4.44)$$

where we first reduce with respect to the first column of multipliers, followed by the second column of multipliers.

Benchmarking

Figure 4.11 shows the execution times for the MATLAB and the GPU implementation of the PLU factorization. In addition, it shows a function that approximates the measured data-points. The function is computed the same way as described in Section 4.1.1, and a subset of the data-points can be viewed in Table B.3.

The two estimated functions are

$$4.11e-2 + 1.97e-10 \cdot x^3$$

and

$$6.10e-2 + 3.32e-10 \cdot x^3$$

for the GPU and CPU respectively. For matrices larger than 800×800 , we can estimate a speedup factor using (4.7). This gives us a factor of 1.69.

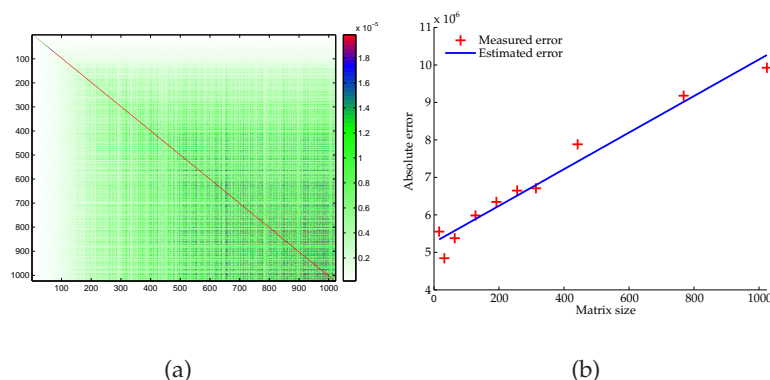


Figure 4.12: Error from computing the PLU factorization of a uniformly distributed random matrix matrix with a large random diagonal. The error is computed as $|PLU - A|$: (a) The mean error of five PLU factorizations. The largest absolute error is on the diagonal, where the elements are largest. The relative error, however, is negligible throughout the matrix. (b) Error of the PLU factorization as a function of matrix size. The estimated function is the best approximation of a first order polynomial in a least squares sense.

Error analysis

The error for the PLU factorization using the Doolittle algorithm is equivalent to the error in the Gaussian elimination, and I refer the reader to Higham [Hig02] for a detailed analysis. Figure 4.12 shows the mean error of five factorizations of full diagonal-dominant random matrices.

The largest error in any of the computations was $3.8586e-5$, and the largest mean error per element of the five computations was $1.0827e-05$. The figure shows a large error on the diagonal, compared to the rest of the matrix. The reason for this is that the diagonal elements are larger than all off diagonal elements. Figure 4.12(b) shows the mean error as a function of the matrix size. The absolute error is minimal, and the relative error is negligible ($< 0.001\%$).

4.2 The GPU as a mathematical coprocessor

Many scientific papers in the field of GPGPU focus on the efficiency of the GPU, showing that it can perform calculations faster than equivalent, but highly tuned algorithms on the CPU. But there has been little, or no, research on utilizing the GPU simultaneously, as a coprocessor to the CPU. This section presents a way of utilizing the GPU and CPU simultaneously

that is easily extendible to multiple GPUs.

4.2.1 Nonblocking calls

The essence of the implementation is nonblocking calls to functions. To define a nonblocking call, a blocking call must be defined first. A blocking call is a call which stalls the processor, waiting for it to finish. A typical example is data IO, which takes a substantial amount of time to complete. While data is read or written, the processor simply idles.

A nonblocking function call is the exact opposite. To use the data IO example, a nonblocking IO call would not stall the processor, but return instantly. This enables the processor to continue processing, while the data is read or written in the background.

Many of the calls to OpenGL are blocking, even though some are non-blocking. Examples of blocking calls include swapping framebuffers and transferring data to and from the GPU without using pixel buffer objects (PBOs). Because we have to execute blocking calls to OpenGL regularly in the presented algorithms, the CPU mostly idles whilst the GPU is processing. To enable the GPU to process in the background through a non-blocking call, the program execution is split into two separate threads.

4.2.2 Threading

A thread is a way for a program to execute two or more simultaneous tasks in parallel. In contrast to using multiple processes, which all own their own resources, threads share resources:

Definition 4.5 ([Wik07c]):

Threads do not own resources except for a stack and a copy of the registers including the program counter.

This is their strength, and their weakness. Because they share resources, we can encounter three thread-specific errors [Kem05a]:

1. Race conditions,
2. Deadlocks,
3. Priority failures.

These three errors have different consequences. The first error leads to unpredictable results, The second will stop program execution, and the third simply slows down the execution rate.

Race condition – A race condition is when two or more threads depend on a shared state. An example is if two threads try to alter a shared variable simultaneously. They will both read the same input value, alter the variable, and write the result. However, because both processes access the same memory at the same time, the values being written or read are arbitrary, resulting in undefined behaviour. This was one of the root software defects of a radiation therapy machine that “massively overdosed six people” [LT93]. But race conditions can be programmed away using a mutual exclusive lock (mutex):

Definition 4.6 ([Kem05b]):

A Mutex object has two states: locked and unlocked.

A mutex is used to lock access to a shared resource several threads use, such as data. By letting each thread wait for access to the resource while it is locked, the possibility of a race condition is eliminated.

Deadlock – The second item, deadlock, is an artifact of the mutex. A deadlock is a situation where thread T_A wants access to resource R_B held by thread T_B . Thread T_B , on the other hand, wants access to resource R_A currently held by T_A , shown in Figure 4.13. Because both threads hold locks on resources the other thread wants, neither can continue. In general, a deadlock will occur when all four Coffmann conditions [CES71] are present:

1. Tasks claim exclusive control of the resources they require (“mutual exclusion” condition).
2. Tasks hold resources already allocated to them while waiting for additional resources (“wait for” condition).
3. Resources cannot be forcibly removed from the tasks holding them until the resources are used to completion (“no preemption” condition).
4. A circular chain of tasks exists, such that each task holds one or more resources that are being requested by the next task in the chain (“circular wait” condition).

Priority failure – Priority failure is the last of the three thread-specific errors.

Definition 4.7 ([Kem05a]):

A priority failure (such as priority inversion or infinite overtaking) occurs when threads are executed in such a sequence that required work is not performed in time to be useful.

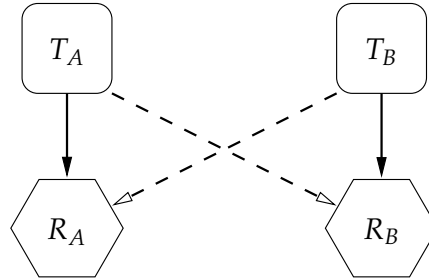


Figure 4.13: A deadlock between thread T_A and T_B , stalling execution. Dotted lines represent requested locks, while solid lines represent acquired locks. The hexagons represent resources.

Starvation is a priority failure where one thread hogs a resource needed by other threads for them to continue processing. In order to prevent one thread from starving all other threads, conditions are used.

Definition 4.8 ([Kem]):

An object of class condition is a synchronization primitive used to cause a thread to wait until a particular shared-data condition (or time) is met.

4.2.3 Multi-threading in single-threaded environment

In order to use multiple threads in our toolkit for MATLAB, we need to ensure that both the MATLAB and OpenGL functions we call are thread-safe:

Definition 4.9 ([Wik07d]):

A subroutine is reentrant, and thus thread-safe, if it only uses variables from the stack, depends only on the arguments passed in, and only calls other subroutines with similar properties.

If the functions are not thread-safe, we cannot arbitrarily call them from different threads.

The 2007a pre-release of MATLAB supports multi-threaded execution of many operations. The MEX API, however, continues to support execution from only one thread [Käl07]. This also implies that all calls to MATLAB have to come from the thread where the `mexFunction` is executed.

The driver implementation of OpenGL from NVIDIA is not thread-safe either. It is up to the hardware vendor to supply thread-safe implementations of OpenGL; even though some thread-safe implementations exist, most implementations are not thread-safe. Because the implementation used in this thesis is not thread-safe, all calls to OpenGL have to come from the thread that created the OpenGL context.

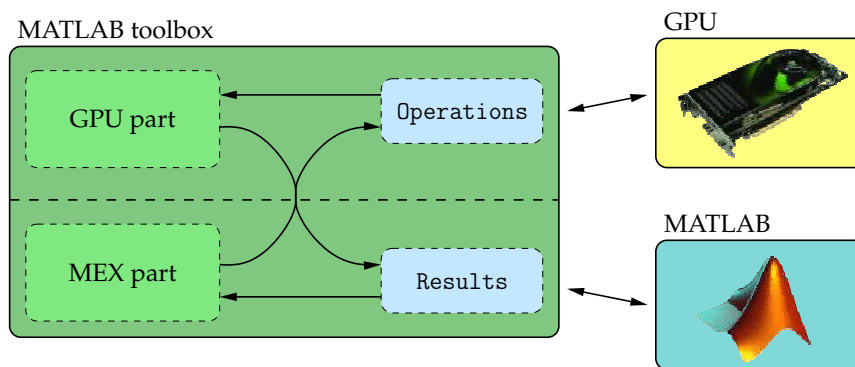


Figure 4.14: Overview of the MATLAB toolbox design. The MATLAB toolbox is run as two separate threads executing in parallel. The MEX part communicates with MATLAB, filling the Operations queue with operations, while the GPU part executes the items in the Operations queue, and saves the results in the Results map. The MEX part can then find the result by looking up the correct operation id in the Result map.

Because of these two requirements, the program logic has to be split into two for multi-threaded execution. Figure 4.14 shows the separation of logic, where one thread communicates with MATLAB, and the other communicates with the GPU. The two parts communicate with each other through a queue of operations, Operations, and a map of results, Results.

Because the two threads share memory, mutexes and conditions are used to prevent race conditions and ensure proper scheduling. It would, theoretically, be best to use a separate mutex per shared resource, but because the time each thread locks the mutex is far less than the execution time of any single operation, I have chosen to use a single mutex.

Figure 4.15 shows a simplified flowchart of the two threads. A lot of intricate details are hidden, but the main flow of execution is shown. When an operation is enqueued, the MEX part adds it to the queue, and returns an id, opId, to MATLAB. The GPU part is then notified, and starts executing the operation. When the operation has completed, the result is stored in the Results map under the key opId. If the result has been requested before it is finished, the MEX part is simply set to wait, until the GPU has completed an operation. It is then notified, and checks whether the result is computed yet. By setting a timeout value for the wait, we ensure that we do not wait for ever. If the timeout is reached, we assume that the operation has failed, and the user is alerted.

This way of programming the operations enable us to return an operation id instantly instead of stalling the CPU throughout the execution of the GPU program. The program automatically translates between the op-

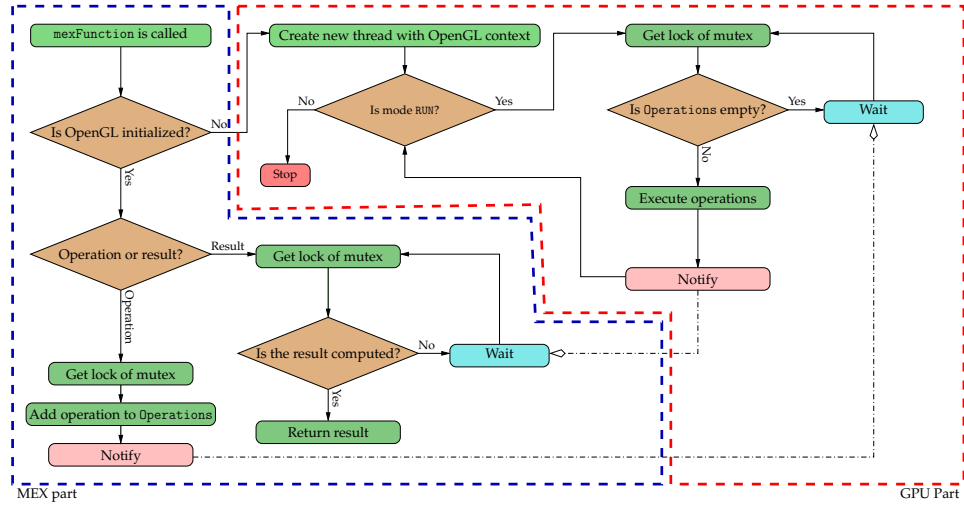


Figure 4.15: Flowchart showing the two threads of execution. The brown diamonds represent if-tests, and the rounded quads represent actions. The dotted lines with diamond arrow represents notifications.

eration id and a matrix, enabling us to enqueue operations of matrices that have not yet been computed. For example, if we want the following operations to be computed on the GPU,

$$\begin{aligned} C &= A * B \\ D &= C^2 \end{aligned} \quad (4.45)$$

we can enqueue both operations even before the result of A, B, and C is known, as shown in Listing 4.7. This enables us to enqueue a series of operations to be executed on the GPU, and lets them execute in parallel as we continue working on the CPU.

4.2.4 Automatically tuned workload distribution

The use of the GPU as a mathematical processor can efficiently speed up computations. For maximum performance, a simple benchmark can be run for different matrix sizes to estimate the ideal load distribution ratio. The following crude approximation gives a pointer to the optimal load ratio between the CPU and the GPU.

Figure 4.16 shows the results of the benchmark for the PLU factorization. The use of both the CPU and GPU simultaneously decreases the total computational time. The load ratio is simply computed as

$$r = \text{ceil} \left(\frac{t_{\text{GPU}}}{t_{\text{CPU}}} \right), \quad (4.46)$$

Listing 4.7: Example MATLAB code that executes in the background

```

1 %Enqueues operation, returns instantly
  A = gpuMatrix(rand(n, n));
  B = gpuMatrix(rand(n, n));

5 %Enqueues the multiplication, returns instantly
  C = A * B;

  %Enqueues the multiplication, returns instantly
  D = C*C;

10 %Enqueues reading back to the CPU, returns instantly
   %Note that this function does not need to be called,
   %but improves performance
   read(D);

15 %Execute instructions on the CPU
   %while the GPU is working simultaneously
   ...

19
20 %Waits for the read operation to complete, and returns
   %the result
   d = single(D);

```

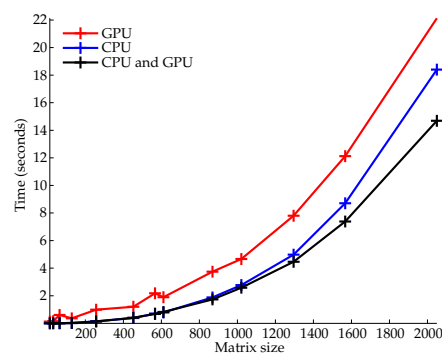


Figure 4.16: The time used to compute 25 consecutive PLU factorizations of systems of size $n \times n$ using the CPU, the GPU, and both simultaneously.

where r is the number of PLU factorizations on the CPU we should perform per PLU factorization on the GPU. The actual computations are executed by queueing one PLU factorization on the GPU, computing r PLU factorizations on the CPU, and finally reading back the result from the GPU. This approach can also be applied to the other algorithms presented in the toolbox, and makes the computations on the GPU virtually free.

Chapter 5

Application

“If I had some duct tape, I could fix that”

— MacGyver

In this section, I will apply the MATLAB toolbox presented in this thesis to a real-world problem, showing its efficiency and usefulness. The application problem is solving the shallow water equations using an implicit discretization, utilizing both the GPU and the CPU in parallel as a heterogeneous processing platform. The load on the GPU and CPU is estimated, yielding good load balance between both processors (as discussed in Subsection 4.2.4).

5.1 The shallow water equations

The shallow water equations is a system of quasilinear hyperbolic partial differential equations [Cas90]. They can describe the evolution of an incompressible fluid due to gravitational acceleration.

The system of equations can be written as

$$\begin{aligned}\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} &= -g \frac{\partial z}{\partial x} - \gamma u \\ \frac{\partial v}{\partial t} + u \frac{\partial v}{\partial x} + v \frac{\partial v}{\partial y} &= -g \frac{\partial z}{\partial y} - \gamma v \\ \frac{\partial z}{\partial t} + \frac{\partial ((h+z)u)}{\partial x} + \frac{\partial ((h+z)v)}{\partial y} &= 0\end{aligned}\tag{5.1}$$

in two dimensions [Cas90]. Here h is the water depth, z is the water elevation, and u and v are the velocities in x and y dimension, respectively. Furthermore, g is the gravitational acceleration, and γ is the bottom friction

coefficient, in this case given by

$$\gamma = g \frac{\sqrt{u^2 + v^2}}{C_z^2 (h + z)}, \quad (5.2)$$

where C_z is the dimensionless Chezy friction coefficient [Kho04].

When solving these equations on a computer, we have to discretize them. This can be done using a finite difference approach to approximate the partial derivatives. There are many different ways of approximating the derivatives explicitly, ranging from low order, to highly accurate higher order schemas. The more accurate a schema is, the more computationally demanding it usually is. High order explicit schemas also impose stability conditions, often related to the velocity at which the wave is propagating. Usually, a Courant-Friedrichs-Lewy (CFL) condition is given where the discretization is stable. The CFL condition implies that we must solve the equations using small time-steps even if we do not have very high velocities. To circumvent this restriction, implicit schemas can be used, and Casulli [Cas90] has presented two implicit finite difference schemas where the stability does not depend on the velocity.

5.1.1 Implicit discretization

Casulli [Cas90] has presented two numerically equivalent approaches to solving the shallow water equations implicitly using a finite difference approach. Both are discretized over a staggered grid, shown in Figure 5.1. The first discretization is a semi-implicit approach, where the velocities are discretized implicitly. The water height and depth are discretized explicitly, and the approach results in a penta-diagonal system of $m \times n$ linear equations of $m \times n$ unknowns.

The second approach is an enhancement to this first approach. Instead of solving both the velocities, u and v , as well as the water elevation in each time-step, Casulli used an alternating direction semi-implicit (ADI) approach. In this approach, the u velocity is solved at times $k + 1/2$, while the v velocity is solved at times $k + 1$. Figure 5.2 shows the relationship between the previously computed values, and the new values we compute.

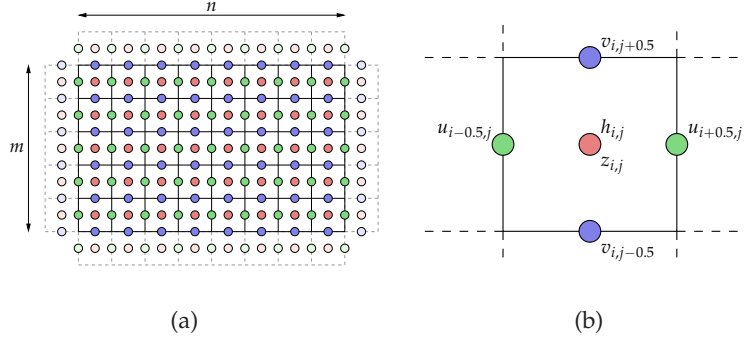


Figure 5.1: The staggered grid used to solve the shallow water equations implicitly: (a) The red dots represent the water elevation, the green dots are the u velocity, and the blue dots represent the v velocity. The dotted cells represent shadow cells used to implement different boundary conditions. (b) The water height is known at the cell midpoints, (i, j) . The u velocity is known at half-steps of i , $(i - 0.5, j)$ and $(i + 0.5, j)$, and the v velocity is known at half-steps of j , $(i, j - 0.5)$ and $(i, j + 0.5)$.

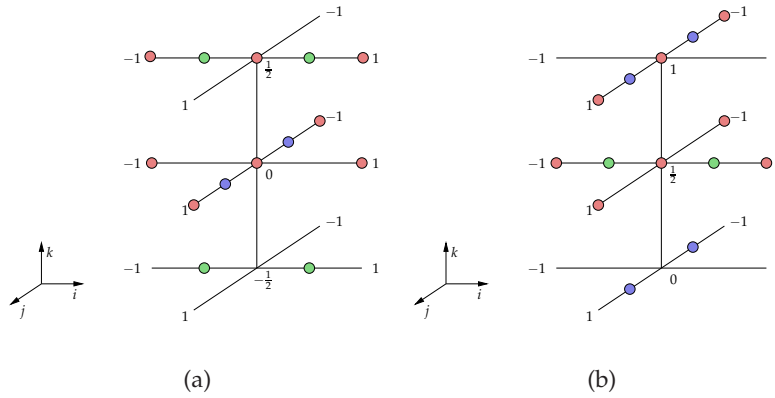


Figure 5.2: The two stencils used to compute a full time-step in the ADI discretization of the shallow water equations. (a) The stencil used in the computation of the first half-step $k + 1/2$ of the water elevation, and the u velocity. (b) The second stencil which is used to compute the next time-step $k + 1$ of the water elevation and the v velocity. Notice that the u and v velocities are updated only in the half and full time-steps, respectively.

The ADI discretization of (5.1) for the first half time-step (solving the u velocity) reads:

$$\left(1 + \Delta t \gamma_{i+1/2,j}^k\right) u_{i+1/2,j}^{k+1/2} = F(u^{k+1/2})_{i+1/2,j} - g \frac{\Delta t}{\Delta x} \left(z_{i+1,j}^{k+1/2} - z_{i,j}^{k+1/2}\right) \quad (5.3)$$

$$\begin{aligned} z_{i,j}^{k+1/2} = & z_{i,j}^k - \frac{\Delta t}{2 \Delta x} \left[\left(z_{i+1/2,j}^k + h_{i+1/2,j}\right) u_{i+1/2,j}^{k+1/2} \right. \\ & \left. - \left(z_{i-1/2,j}^k + h_{i-1/2,j}\right) u_{i-1/2,j}^{k+1/2} \right] \\ & - \frac{\Delta t}{2 \Delta y} \left[\left(z_{i,j+1/2}^k + h_{i,j+1/2}\right) v_{i,j+1/2}^{k+1/2} \right. \\ & \left. - \left(z_{i,j-1/2}^k + h_{i,j-1/2}\right) v_{i,j-1/2}^{k+1/2} \right]. \end{aligned} \quad (5.4)$$

Eliminating $u^{k+1/2}$ in (5.4) and restructuring gives

$$\begin{aligned} -\mu \alpha_{i-\frac{1}{2},j}^k z_{i-1,j}^{k+\frac{1}{2}} + \left(1 + \mu \alpha_{i+\frac{1}{2},j}^k + \mu \alpha_{i-\frac{1}{2},j}^k\right) z_{i,j}^{k+\frac{1}{2}} - \mu \alpha_{i+1,j}^k z_{i+\frac{1}{2},j}^{k+\frac{1}{2}} = \\ z_{i,j}^k - \rho \left[\beta_{i,j+\frac{1}{2}}^k v_{i,j+\frac{1}{2}}^k - \beta_{i,j-\frac{1}{2}}^k v_{i,j-\frac{1}{2}}^k \right] \\ - \sigma \left[\alpha_{i+\frac{1}{2},j}^k \left(F(u^{k-\frac{1}{2}})_{i+\frac{1}{2},j}\right) - \alpha_{i-\frac{1}{2},j}^k \left(F(u^{k-\frac{1}{2}})_{i-\frac{1}{2},j}\right) \right], \end{aligned} \quad (5.5)$$

where

$$\begin{aligned} \mu &= g \frac{\Delta t^2}{2 \Delta x^2}, & \sigma &= g \frac{\Delta t}{2 \Delta x}, & \rho &= g \frac{\Delta t}{2 \Delta y}, \\ \gamma_{i,j}^k &= \frac{g \sqrt{u_{i,j}^{k^2} + v_{i,j}^{k^2}}}{C_z^2 (h_{i,j}^k z_{i,j}^k)}, & \alpha_{i,j}^k &= \frac{z_{i,j}^k + h_{i,j}^k}{1 + \Delta t \gamma_{i,j}^k}, & \beta_{i,j}^k &= z_{i,j}^k + h_{i,j}^k. \end{aligned} \quad (5.6)$$

Because we need the water elevation z and depth h at the intersection between cells in (5.5), we use a simple algebraic mean of the two closest cells.

The operator $F(w)$ is a nonlinear finite difference operator that represents the convective terms in (5.1). For numerical stability, $F(w)_{i,j}$ integrates a streamline through the velocity field, from (i, j) to (i', j') . $F(w)_{i,j}$ is the value of the property w at the point (i', j') (see Figure 5.3).

The equations for the second half time-step (solving the v velocities) are similar, and I refer to Casulli [Cas90] for a detailed derivation.

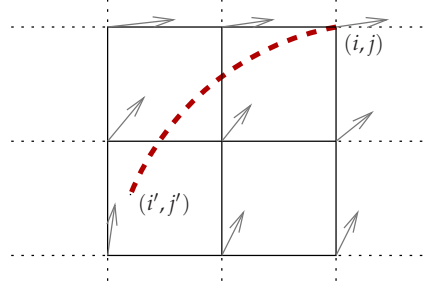


Figure 5.3: Streamline integration through the velocity field, (u, v) , to calculate $F(w)$. The red dotted line is the streamline which is integrated backwards one time-step originating from the point (i, j) .

5.1.2 Setting up the linear system

Equation (5.5) represents a tridiagonal system of n equations in n unknowns for each j :

$$\begin{bmatrix} b_1 & -a_{\frac{1}{2}} & 0 & 0 & 0 & \cdots & 0 \\ -a_{\frac{1}{2}} & b_2 & -a_{\frac{3}{2}} & 0 & 0 & \cdots & 0 \\ 0 & -a_{\frac{3}{2}} & b_3 & -a_{\frac{5}{2}} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & \cdots & 0 & -a_{n-\frac{5}{2}} & b_{n-2} & -a_{n-\frac{3}{2}} & 0 \\ 0 & \cdots & 0 & 0 & -a_{n-\frac{3}{2}} & b_{n-1} & -a_{n-\frac{1}{2}} \\ 0 & \cdots & 0 & 0 & 0 & a_{n-\frac{1}{2}} & b_n \end{bmatrix} \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ \vdots \\ z_{n-2} \\ z_{n-1} \\ z_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_{n-2} \\ c_{n-1} \\ c_n \end{bmatrix} \quad (5.7)$$

where

$$a_i = \mu \alpha_{i,j}^k, \quad b_i = 1 + \mu(\alpha_{i+\frac{1}{2},j}^k + \alpha_{i-\frac{1}{2},j}^k),$$

and c_i is the right hand side of (5.5). This system is positive definite [Cas90], and thus easily solvable with a unique solution.

The observant reader will notice the first and last equations in (5.7) do not fulfill the discretized (5.5). These are the equations we modify to suit our boundary conditions. In our model, we enforce Dirichlet boundary conditions on the u and v velocities by setting them to zero in the normal direction. By examining (5.3) we see that the normal velocity will equal zero if the water height at the boundary and the shadow cell are equal:

$$z_{i,j} = z_{i+1,j}, \quad u_{i+\frac{1}{2},j}^{-\frac{1}{2}} = 0 \quad \Leftrightarrow \quad u_{i+\frac{1}{2},j}^k = 0 \quad \forall k \geq 0, \quad i = \{0, n\}. \quad (5.8)$$

Thus, we set the normal velocity to zero by setting the water elevation at the boundary cells to be equal the closest cell within our computational domain.

5.2 Implementation

Solving the shallow water equations using the implicit discretization presented in the previous section ultimately boils down to setting up the linear systems, and solving them. The linear systems (5.7) can be computed explicitly, and solved using the PLU factorization. It should be mentioned that they can also be solved far more efficiently, by storing only the three diagonal-vectors, and using Gaussian elimination for tridiagonal matrices, or using a preconditioned conjugate gradients solver [Cas90]. Nevertheless, the linear systems are solved using the PLU factorization presented here to demonstrate its usefulness.

5.2.1 MATLAB reference implementation

The MATLAB reference implementation is fairly straight forward. In each time-step, we have two loops where a system is set up, and then solved. In the first loop, we set up the tridiagonal system (5.7) for each j . The system is solved using the internal PLU factorization in MATLAB. When all the systems have been solved, the u velocity is solved explicitly using (5.3). In the second loop, we set up the equations for the second half-step of the ADI discretization, and again solve using the PLU factorization in MATLAB. The v velocity is also solved explicitly.

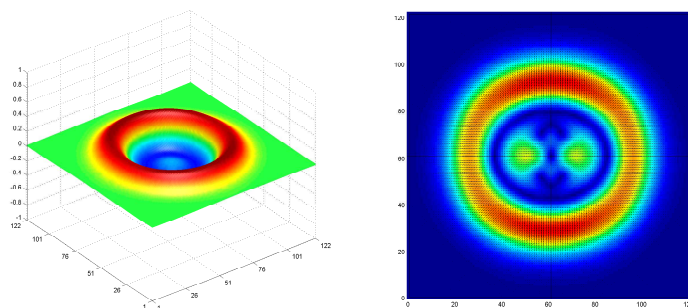
The nonlinear finite difference operator $F(w)$ is in this implementation chosen to use bilinear interpolation of the underlying discrete velocity field. The integration is computed using a backward Euler approach, where the velocity field is assumed to be constant in time. Listing C.1 shows part of the code used to both set up the equations, and solve the system.

5.2.2 MATLAB implementation using the GPU toolbox

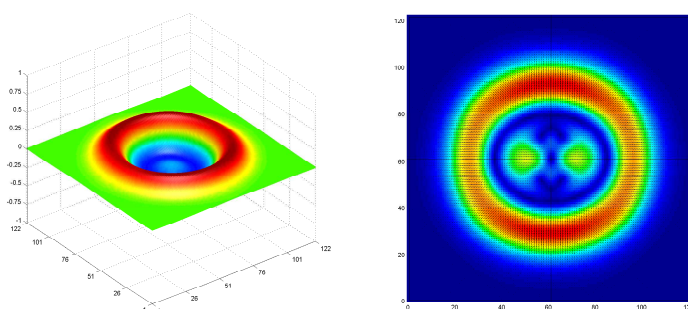
The MATLAB implementation that uses the GPU toolbox is almost identical to the reference implementation. But in contrast to the reference implementation, the linear systems are here solved using the PLU factorization on the GPU *in addition* to solving using the MATLAB internal implementation. The load distribution is determined in the same manner as described in subsection 4.2.4

5.3 Results

The speedup from using the toolbox for the problem size shown here is minimal. The problem size is here 121×121 , as shown in Figure 5.4. Larger problems, however, will benefit from using the GPU, and experience speedups similar to what is shown in subsection 4.2.4.



(a)



(b)

Figure 5.4: The shallow water equations solved numerically. The initial conditions create a wave propagating slightly faster in one direction: (a) shows the wave, and the velocity field at time $t = 10$ solved using the CPU alone. (b) shows the wave, and the velocity field at time $t = 10$ solved using the GPU *and* the CPU simultaneously.

Table 5.1: Table over the absolute maximum and mean error between the CPU and the hybrid approach after ten time-steps.

Error	z	u	v
Mean	1.47e-07	3.82e-10	2.85e-10
Max	4.48e-08	1.10e-06	1.13e-06

Table 5.1 shows the measured error between the CPU and the hybrid (CPU and GPU simultaneously) approach. The mean error is minimal, and the maximum error is also small. The relative measured error was negligible ($< 0.001\%$)

Chapter 6

High-level languages

“There is more than one way to do it”

— Perl motto

There are several high-level programming languages for GPGPU, where the main ones are Close to The Metal (CTM) [Adv06] and Computer Unified Device Architecture (CUDA) [NVI07b], PeakStream [Pea06] and RapidMind [MD06], and Brook [BFH⁺04] and Sh [MDP⁺04]. The first two, CTM and CUDA, are vendor-specific languages developed by ATI and NVIDIA, respectively. They both access the hardware without going through a graphics API. CTM is quasi-assembly, while CUDA is a C++ API *and* a quasi-assembly language. The next two languages, PeakStream and RapidMind are commercial programming APIs for C++. The final two languages, Brook and Sh are opensource APIs for C++. However, their development is minimal, and neither have reached a final release. The following sections review a subset of the mentioned languages; the PeakStream Platform, the RapidMind Development Platform and CUDA. Then, the languages are compared against the toolbox presented in the previous chapter.

6.1 PeakStream

PeakStream [Pea06] is a high-level language that abstracts away many of the intricate details of GPGPU programming. It is implemented as a virtual machine, similar to Java, acting as a layer between the application and the *stream processor*. They do not only aim at running code on GPUs, but also on other stream processors such as the IBM Cell Broadband Engine Architecture (Cell BE) [Hof05]. The advantage of such an approach is that the same source code can be used on all supported platforms. In addition, low-level optimizations are done by the virtual machine, and not by the application. This allows the same code to run faster when new optimizations appear in

the virtual machine.

The virtual machine is accessed via a C/C++ interface by including the PeakStream header file. The primary data-type is array, with support for both single and double precision. Double precision is currently emulated by using the CPU, and they report more accurate single precision computations than what the current hardware allows [Pea06]. The arrays are explicitly moved to and from the GPU by the user as shown in Listing 6.1.

6.2 RapidMind

RapidMind is a commercial spin-off of Sh [MDP⁺04], and their basic data-type is the array. Their API implements several different back-ends, including support for both the GPU and the Cell BE [Hof05], as well as a CPU back-end. The programmer can actively choose which back-end to use, or let the API use the most efficient back-end automatically.

To use the RapidMind Development Platform, the user has to write *programs* that operate on the arrays. The *program* is executed on the back-end platform, but written in the C++ source code. Listing 6.2 shows such a program embedded in the C++ code. The program, `triple` is defined to take an input value, `a` of type `Value3f`, multiply it by three, and store the result in `b`. The `Value3f` signifies that each element holds three float values.

Higher-level functions, such as PLU factorization, are not included, and must be written by the user.

6.3 CUDA

The CUDA [NVI07b] library from NVIDIA comes with CUBLAS. CUBLAS is an approach to implementing the BLAS library on the GPU. It is only available the NVIDIA 8800 series GPUs, and in single precision. Instead of accessing the hardware via OpenGL, it is accessed more directly. However, because only a subset of BLAS' functionality is implemented, functions such as PLU decomposition are unavailable.

Listing 6.3 shows the use of CUBLAS to compute the matrix multiplication of two matrices, `a_CPU` and `b_CPU`. The CUBLAS function `cublasSgemm` performs the actual computation, and the user transfers the arrays between the CPU and the GPU explicitly.

6.4 Comparison

The three discussed languages, PeakStream, RapidMind and CUDA, all abstract away the graphical details needed to program the GPU. Because they

Listing 6.1: Source-code for the PeakStream virtual machine to solve the linear system $Ax = b$ using PLU factorization.

```

1 init(NULL); //Init PeakStream VM

    float A_CPU = new float[n*n];
    float b_CPU = new float[n];
5 for (int i=0; i < n*n; ++i) {
    A_CPU[i] = rand() / static_cast<float>(RAND_MAX);
}
    for (int i=0; i < n; ++i) {
    b_CPU[i] = rand() / static_cast<float>(RAND_MAX);
10 }

    float* x_CPU = new float[n*n];
    Arrayf32 x_GPU;

15 { //Begin lifetime of temporary PeakStream variables
    Arrayf32 lu;
    Arrayf32 pivot;
    Arrayf32 singularity;

    20 //Transfer to the GPU
    A_GPU = make2(n, n, A_CPU);
    b_GPU = make1(n, b_CPU);

    //Factorize the matrix
25 lu_decomp(A_GPU, lu, pivot, singularity);

    //Solve x
    x_GPU = lu_solve(lu, pivot, b_GPU);
}
30 //Read back to the CPU
x_GPU.read2(x_CPU, n*n*sizeof(float));

shutdown(); //Shutdown PeakStream VM

```

Listing 6.2: Example source code that uses the RapidMind Development Platform.

```

1 // Initialize RapidMind
  rapidmind::init();

  Program triple = RM_BEGIN {
5   In<Value3f> a; // Input to function
    Out<Value3f> b; // Output of function

    b = 3 * a;
  } RM_END;
10
  //Create original array and output array
  Array<1,Value3f> one_GPU(n);
  Array<1,Value3f> three_GPU;

15 //Set data of GPU array
  float* one_CPU = one_GPU.write_data();
  for (int i=0; i < n * 3; ++i) {
    one_CPU[i] = rand() / static_cast<float>(RAND_MAX);
  }
20
  //Run the program on the array
  three_GPU = triple(one_GPU);

  //Read result back to CPU
25 float* three_CPU = three_GPU.read_data();

```

Listing 6.3: Example source code that uses the CUBLAS library from NVIDIA.

```
1 //Initialize CUBLAS
  cublasInit();

  float* a_GPU;
5 float* b_GPU;
  float* c_GPU;

  //Allocate on CPU, and fill with data
  float* a_CPU = new float[n*n];
10 float* b_CPU = new float[n*n];
  float* c_CPU = new float[n*n];
  for (int i = 0; i < n*n; ++i) {
    a_CPU[i] = rand() / static_cast<float>(RAND_MAX);
    b_CPU[i] = rand() / static_cast<float>(RAND_MAX);
15 }

  //Allocate on GPU
  cublasAlloc(n*n, sizeof(float), &a_GPU);
  cublasAlloc(n*n, sizeof(float), &b_GPU);
20

  //Set data on GPU
  cublasSetVector(n*n, sizeof(float), a_CPU, 1, a_GPU, 1);
  cublasSetVector(n*n, sizeof(float), b_CPU, 1, b_GPU, 1);

25 //Compute matrix multiplication using sgemm
  cublasSgemm('n', 'n', n, n, n, 1, a_GPU, n, b_GPU, n, 1, c_GPU, n);

  //Read back to CPU
  cublasGetVector(n*n, sizeof(float), c_GPU, 1, c_CPU, 1);
30

  //Release CUBLAS resources
  cublasShutdown();
```

are all operating on the same architecture, they employ the same programming paradigms. Nevertheless, they all also present somewhat different approaches.

PeakStream offers a virtual machine, which is binary compatible between versions. In addition, they supply a subset of the BLAS library that executes on the GPU, as well as some higher level functions, i.e., PLU factorization and convolution.

RapidMind has a slightly lower level API that enables the user to write their own programs that operate on the data. This enables the user more control, but also requires more knowledge of the underlying hardware.

NVIDIA has direct access to the underlying hardware with their CUDA library, enabling them to write optimally tuned algorithms. However, the available API is quite different from PeakStream and RapidMind, and is modeled very close to the underlying hardware. It requires a lot of insight into the hardware for optimal efficiency.

The toolbox presented in this thesis, however, represents a totally different approach. It is far more high-level than the others, where a user is blissfully ignorant of the underlying logic and hardware.

However, comparing these four approaches to using the GPU as a computational resource is like comparing apples and pears. RapidMind and PeakStream are aimed at high performance computing, typically on a cluster of nodes. CUDA, on the other hand is a low-level API that accesses the hardware without going through the graphics API. The MATLAB toolbox is an approach far from these existing APIs. It offers the use of the GPU using familiar syntax in a familiar development environment, and requires no prior knowledge of the GPU.

The natural choice for a user familiar with MATLAB will be the toolbox presented here. However, high performance computing applications require more control and efficiency than this toolbox can provide, and are better off using another alternative.

Conclusions

“If something’s hard to do, then it is not worth doing”
— Homer Simpson

7.1 Conclusions

The three questions posed in the introduction,

1. Is it possible to create a toolbox for MATLAB that transparently uses the GPU as a mathematical coprocessor?
2. Is the toolbox accurate enough for use in high-performance applications?
3. Is such an approach competitive, compared with other high-level interfaces to the GPU?

have all been reasonably well answered in this text.

1. The first question is ultimately covered in Section 5.2, where the difference between using the built-in MATLAB functions and the GPU toolbox are shown to be minimal. A user familiar with MATLAB syntax only needs to alter a small number of code lines to benefit from the speed increase offered by the GPU.

2. The second question is covered in Sections 4.1.1, 4.1.2 and 4.1.3 for the three algorithms matrix multiplication, Gauss-Jordan elimination, and PLU factorization. All the presented algorithms have shown to be sufficiently accurate for most purposes. In addition, it is shown that the PLU factorization accurate enough to solve the shallow water equations in Chapter 5.

The emergence of new hardware, e.g., as pronounced by NVIDIA, will eliminate the errors shown in this thesis, making the GPU as accurate as the CPU. Nevertheless, problems related to NaN, INF, etc. might still exist.

3. The third and final question is discussed in Chapter 6, where the toolbox is compared to other programming languages for GPGPU, and shown to be easy to use compared to these. But because the programming languages are not directly comparable it is impossible to judge which is better. While the MATLAB toolbox will be the natural choice for a MATLAB user, applications where high performance is vital will be better off using another alternative where more control of the hardware is given.

7.2 Summary of Contributions

In this thesis, I have presented an interface from MATLAB to the GPU, enabling the use of both the CPU and the GPU simultaneously. I have further presented how to use this interface for full matrix matrix multiplication, Gauss-Jordan elimination, and PLU factorization. I have also shown that these operations are sufficiently accurate, even on today's limited single precision hardware.

I have also presented a new way of computing both Gauss-Jordan elimination, and PLU factorization. The novelties include packing two-by-two sub-matrices into a single texel, as well as a new pivoting strategy.

7.3 Future Research

As all masters theses, this thesis is also limited both in time and scope. As a consequence, some related research topics I find interesting are not covered here. The following, however, is a set of topics closely related to the work presented here.

Automatically tuned linear algebra on the GPU – Even though I have presented a crude approximation to utilizing both the CPU and the GPU for maximum performance, improving this to a fully automatic workload distribution will be of great interest. In addition, tuning the linear algebra algorithms themselves to the underlying hardware, as presented by Jiang and Snir [JS05] will further increase the efficiency.

Utilizing the GPU memory efficiently – Modern GPUs have a lot of texture memory, typically ranging from 512 - 768MB, and new generations will have even more. For memory demanding algorithms, this memory can both be used for calculation, as well as to offload main memory load.

By moving part of the algorithm to the GPU, the overall load on system memory can be decreased, enabling larger systems to be solved.

High-level language bindings – The use of the GPU together with MATLAB has been shown in this thesis. However, other high-level languages such as Python, will also benefit from utilizing the GPU in a similar fashion.

Appendix A

PLU factorization on a cluster of GPUs

The parallel PLU project is a cooperation between Martin Lilleng Sætra, Trygve Fladby and myself. The original idea was to run the application at The Gathering (TG) [KAN05], the worlds largest computer party. Time constraints, and other issues (e.g. security), prevented us from reaching our original goal. We have completed the project, despite that we were unable to run the application at TG, and reported our findings in the following white-paper.

My main contributions, in this project, have been to the design of the overall global algorithm, data communication and the local operations normalize and reduce.

PLU FACTORIZATION ON A CLUSTER OF GPUS USING FAST ETHERNET

André Rigland Brodtkorb, Martin Lilleng Sætra and Trygve Fladby
1st May 2007

Abstract In this white paper, we present a novel approach to solve linear systems of equations on a cluster using the PLU factorization. We use the graphics processing unit (GPU) as the main computational engine at each node, and a block-cyclic data distribution to solve the system. The local computation is a new way of solving the PLU factorization on the GPU. It utilizes the full four-way vectorized arithmetic found in most GPUs, and a new pivoting strategy. The global algorithm uses the message passing interface (MPI) for communication between nodes. We show that our algorithm is highly efficient on the local nodes, but bounded by the relatively slow network. A faster network will eliminate this bottleneck, and the speed of the local computations show promising results.

A.1 Introduction

This paper explores the field of general purpose computation on graphics processing units (GPGPU). We specifically target the PLU factorization of a large system of linear equations on a cluster of nodes. Solving large linear systems of equations using dense algorithms is used extensively as a benchmark for clusters and supercomputers. The High Performance LINPACK benchmark [PWDC] (HPL) which computes the PLU factorization, is the standard way of benchmarking and ranking the fastest 500 supercomputers in the world [UUN]. This benchmark, however, has been criticized for neglecting the importance of faster inter-node communication. This is because the HPL benchmark can run the benchmark with different parameters that compensate for slow network communication by letting each node execute extra computations (e.g., look-ahead).

While the HPL benchmark uses the CPU to compute partial results on each node, we utilize the graphics processing unit (GPU) as the main computational engine to solve the same problem. The GPU is a massively parallel processor with vast amounts of processing power [OLG⁺07]. Current GPUs have a theoretical peak of 400 GFLOPS [Neo07], compared to 90 GFLOPS [Neo07] for current high-end CPUs. When comparing the price¹ per FLOP, the GPU comes out ahead as well with approximately \$1.50 per GFLOP, compared to the CPU that costs approximately \$18 per GFLOP.

During the last years, we have seen an enormous development in 3D-graphics. The demand for more powerful programmable graphics processing units (GPU) from for example the gaming industry has led to an increased flexibility in the processors. The rapid evolution in speed and flexibility has made the GPU interesting for scientific purposes as well. The field of general-purpose computation on GPUs (GPGPU) has emerged as a new and exiting research area [OLG⁺07]. Even though the GPU is a far more powerful and cost-effective processor than the CPU, there is another price. While the CPU has complex logic for branch prediction, cache management, and instruction pipelining, most of the transistors on the GPU are used for pure floating-point operations. There is another architectural difference as well. The CPU is designed to operate on sequential code, such as word processing where each character is entered and processed sequentially. The GPU

¹Prices are from the Norwegian web shop komplett.no 2007-04-23.

on the other hand, is designed to simultaneously compute all the pixels that together make up the screen image. In addition, the GPU could traditionally only be accessed via a graphics API, such as OpenGL [Khr07] or DirectX [Mic07b]. The architectural differences, and the need to access the GPU through a graphics API require new algorithms and techniques to be employed when the GPU is to be used for general-purpose computing.

A.2 Background

The Top 500 project [UUN] was started in 1993 to provide a reliable basis for tracking and detecting trends in the field of high-performance computing. It is a list of the 500 most powerful supercomputers, which is updated twice per year. The ranking of the supercomputer sites is determined by how well they perform on the LINPACK benchmark. A parallel version of LINPACK named HPL [PWDC] was introduced by J. Dongarra, for this purpose. HPL is short for High-Performance LINPACK Benchmark for Distributed-Memory Computers. HPL utilizes the Message Passing Interface (MPI) and the Basic Linear Algebra Subprograms (BLAS). The algorithm used by HPL implements a two-dimensional block-cyclic data distribution. In addition a look-ahead strategy and bandwidth reducing swap-broadcast algorithm is used to increase performance. The complete operation count sums up to $\mathcal{O}(\frac{2}{3}n^3) + \mathcal{O}(n^2)$.

LU factorization on the GPU has previously been implemented by Galoppo et al. [GGHM05]. One of their main contributions was index-pair streaming, which uses texture coordinates to make a cache-oblivious algorithm. The index-pair streaming technique sets texture coordinates from the CPU in order for the GPU to pre-fetch data, in contrast to computing them on the fly on the GPU. This data pre-fetch resulted in about 25% speed increase [GGHM05]. They also reported their algorithm as faster than ATLAS, but the benchmark was highly synthetic.

To run our application in parallel on multiple nodes, we have utilized the Message Passing Interface 2.0 (MPI-2) [MPI]. MPI-2 is a C/C++ and Fortran interface for message passing between multiple processes spread over any number of nodes. It can be used in many different setups, e.g., supercomputers, distributed memory clusters, and shared memory clusters. Several implementations of MPI-2 exist, where we have chosen MPICH2 [Arg] for our application. The most important uses of MPI-2 in our application are the automatic generation of a block-cyclic Cartesian grid of processes and broadcast of data to groups of processes.

There are two concepts related to our use of MPI-2 that require some explanation; communicators, and blocking- and non-blocking calls. A *communicator* in MPI is a collection of processes. Many functions in MPI-2 take a communicator as argument and perform the requested operation on all processes in that communicator. A call to the broadcast function in MPI, for example, can look like this: `MPI_Bcast(buf, 10, MPI_FLOAT, 0, MPI_COMM_WORLD)`. This call will broadcast ten elements of the array `buf` to all processes in the `MPI_COMM_WORLD` communicator. The other processes in the communicator must also call the `MPI_Bcast` function to receive these elements. The `MPI_COMM_WORLD` communicator is a special communicator that contains all processes, and it is initialized automatically by MPI. When an MPI function is called on all processes within a communicator (or group) it is referred to as a collective operation. `MPI_Bcast` is a collective operation.

A *blocking* call will make the application wait for the call to complete before continuing execution. In this way you will know if the call has finished successfully or aborted due to some error. This also means that the application may get

Listing A.1: Example on a deadlock in an MPI-2 program

```
1 MPI_Init(&argc, &argv);

    if(processId == 0) {
        MPI_Recv(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD, &status);
5    MPI_Send(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD);
    } else(processId == 1) {
        MPI_Recv(buf, 10, MPI_INT, 0, 100, MPI_COMM_WORLD, &status);
        MPI_Send(buf, 10, MPI_INT, 1, 101, MPI_COMM_WORLD);
    }
10 MPI_Finalize();
```

deadlocked, where two or more processes have called competing blocking functions that are circularly dependent on each other [CES71]. For example, if we have two processes that execute the code in Listing A.1, it will result in a deadlock. Both processes are waiting for the other to send data, thus blocking program execution. A non-blocking call on the other hand, will not cause the application to wait for the call to return. In this way it is possible to call a function and continue executing the application before the function returns. Collective operations in MPI-2 are always blocking.

A.3 Algorithm

The LU factorization of a matrix A can be written as $LU = A$, where L and U are *lower* and *upper* triangular respectively. Using the Doolittle algorithm, we can construct the upper triangular matrix U using Gaussian elimination. The lower triangular matrix is constructed from the multipliers used to reduce A to an upper triangular form. For our algorithm to be numerically stable, we also permute the rows of A . This is known as partial pivoting, and ensures that the row we are eliminating with creates smaller perturbations of the result than would normally occur. With the permutation of the rows in A , our factorization takes the form $A = P^T LU$, where P is the permutation matrix that permutes rows of A .

Our algorithm has two layers, the global and the local computation. The global algorithm solves the PLU factorization of the matrix spread over all the nodes, shown in Figure A.1(b), whilst the local algorithm is what each node needs to compute for the global algorithm to be correct.

Each node in the computation receives a block-cyclic part of the matrix, as shown in Figure A.1(b). Then, all the processors compute what type of operation they need to compute. Our algorithm splits the computation into four distinct operations: pivot, normalize, eliminate and reduce, as shown in Figure A.1(a). The operation computed on each node depends on the global position of the pivot operation. All processors that hold elements in the same row as the pivot operation need to compute the normalize operation, and similarly all nodes with elements in the same column as the pivot operation need to compute the eliminate operation. All remaining nodes need to compute the reduction operation. In Figure A.1(b) this means that process 0 is the *pivot*, process 1 executes *normalize*, process 2 *eliminate*, and process 3 *reduce*. The pivot node shifts one down along the diagonal for

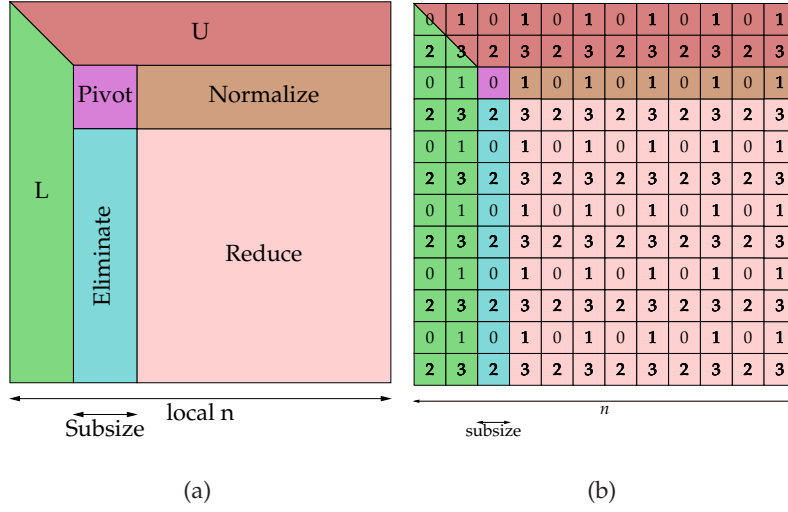


Figure A.1: PLU decomposition on a cluster of nodes: (a) the four different parts of the LU factorization. (b) the block-cyclic distribution of data on four nodes, 0, 1, 2 and 3.

each global pass.

A.3.1 Global algorithm

Computing the PLU factorization is an almost embarrassingly parallel operation. However, vanilla implementations demand a lot of data to be transferred between nodes, which is a very costly operation. In addition, many nodes would simply idle as we reach the end of the computation.

To reduce the idling, we distribute the matrix A block cyclically in the same fashion as the HPL algorithm [PWDC]. Figure A.1(b) shows this distribution, where all nodes have a part of the matrix to process throughout the whole factorization, except for the very last block. The last block is computed by the last node in an extra pass. For each pass in the global domain, we compute the result of one row of blocks, and one column of blocks. In the following, we refer to these as *block-row* and *block-column* respectively.

To lessen the amount and number of transfers between nodes, we use partial pivoting within in-core memory, thus eliminating the need to transfer rows between processors. It is trivial to create examples where partial pivoting fails, but sufficient accuracy is attainable in practice. This also holds for our pivoting, which pivots in a subset of the regular pivot candidates.

In order to compute one pass in the global domain, we have to execute the four different operations *pivot*, *normalize*, *eliminate* and *reduce*. It should be mentioned that this data distribution, and splitting into different operations per node allows for multiple nodes, not only four as shown in this example. In the third pass of this algorithm, we have the following situation (see also Figure A.2):

Pivot: The pivot position (process 0) must compute the PLU factorization of the current active pivot block in its local domain. The block size is $\text{subsize} \times$

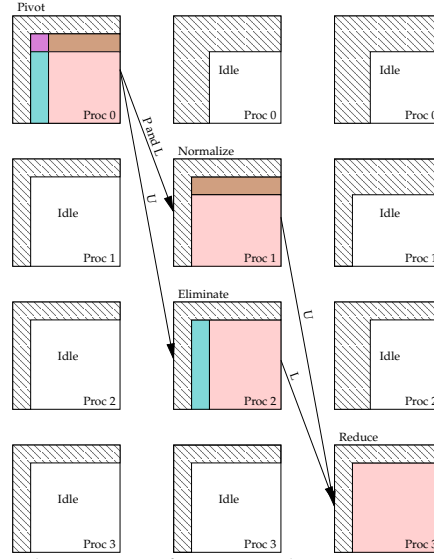


Figure A.2: Data send patterns for PLU decomposition using four nodes in the third global pass (corresponds to the situation in Figure A.1(b)). The shaded areas represent the part of the matrix we already have computed.

subsize. In addition, it has to reduce the rest of the local matrix according to the computed L and U . These blocks belong elsewhere in the global domain (see Figure A.1(b)). In each global pass, there is always only one pivot node.

Normalize: The normalize operation (process 1) needs to compute U according to the P and L computed by the *pivot* operation. It will also have to reduce all remaining elements in the local matrix, which again belong elsewhere in the global domain. There are $s - 1$ nodes that compute the normalize operation in each global pass, where s is the width and height of the processor grid.

Eliminate: Eliminate (process 2) calculates the multipliers needed to forward substitute one block by using the computed U 's from *pivot*. In addition, it has to reduce the rest of the local matrix, according to the computed U . In each global pass, the number of eliminate nodes is also $s - 1$.

Reduce: The reduce operation simply reduces the local matrix according to the L and U computed in *eliminate* and *normalize* respectively. All remaining processes compute this operation, $s \times s - 2(s - 1) - 1$ nodes.

As stated in the list of operations, the different processes depend on data from other processes. This dependency is not static, but varies with the operation the current node is set to execute. Figure A.2 shows how the data is sent in the already used example. The nodes waiting for data cannot continue before they have received the data. This effectively limits the computational speed to the slowest node. The HPL [PWDC] algorithm uses look-ahead to remedy this somewhat. As this chart shows, there is still quite a lot of idling for the four nodes. The pivot node, for example, computes its result and then waits until all other nodes have completed their computations.

A.3.2 Local algorithm

The local algorithm includes four stages *pivot*, *eliminate*, *normalize* and *reduce*, but first we will introduce the matrix representation. The data is row-wise represented in four-wide vectors [Mor03]. This is to utilize as much computational power and bandwidth as possible, since most GPUs can execute one MAD instruction on four-long vectors per clock cycle. The advantage of this packing scheme is that it does not require restructuring of the data in main memory before it is sent to the GPU². Another reason for this choice is that it fits well with the solution we have for pivoting. In addition to storing the matrix, we add an extra column leftmost in the matrix, as shown in in Figure A.3(a). This column is used to speed up the calculation of the next pivot element, explained later. Because the result of writing to the same buffer as we read from is explicitly undefined in OpenGL, we have to use an extra texture. The two textures are used as one virtual matrix, but we alternate between reading / writing and writing / reading to the front and back textures, respectively. This technique is referred to as ping-ponging in the field of GPGPU.

Pivot

The pivot procedure computes the PLU factorization of A , but stops when one block-row and one block-column has been computed (see Figure A.1(b)). It can roughly be split into two tasks: multiplier calculation, and reduction, each explained below. To compute a single row and column, we start by permuting the first column simultaneously as we compute the multipliers. Then, we reduce the rest of the matrix, whilst permuting the rows here as well.

To compute one column of multipliers, we read from the correct location in the source texture, and write to the leftmost column in the destination, as shown in Figure A.3(a). The top element is rendered at the position of the pivot element. Because the multiplier for the top row always is one, we do not need to compute it. In addition to computing the multipliers, we also compute the values of the column to the right of the pivot position and store in one of the other color channels (see Figure A.3(b)).

When the computation is complete, we transfer the multipliers and the reduced next column to the CPU using a pixel buffer object (PBO). The PBO uses asynchronous read-back to the CPU, allowing both the CPU and the GPU to continue execution. When the whole leftmost column has been transferred to the CPU, the next pivot element is found by the CPU. Simultaneously as the data is copied, and the CPU searches for the pivot element, the GPU subtracts the multiplier times the top row throughout the rest of the matrix. The top and pivot row are also interchanged simultaneously in the same manner as in the first column. In addition, we employ the index pair streaming technique to increase performance [GGHM05]. When the computation is complete, the top row is copied to the CPU again using a PBO. The algorithm continues until we have computed the whole block-row of U , and block-column of L .

Normalize

The normalize step computed on the local domain executes as follows. The L matrix from this global time-step's pivot node is uploaded to the GPU as a texture. Then, we execute a for-loop that sequentially computes one row of U at a time.

²Assuming its width is divisible by four.

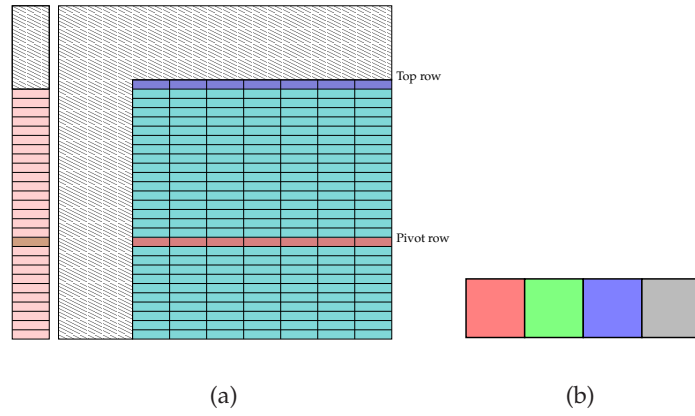


Figure A.3: Data representation on the GPU: **(a)** Row interchange of the multipliers (leftmost column) and the rest of the matrix (cyan part). The magenta element is the current pivot position, and is swapped with the brown element, the current maxim in the pivot column. The red and blue rows in the rest of the matrix are the rows corresponding to the brown and magenta elements respectively. **(b)** The leftmost column of the texture, with both the multiplier, and the reduced next column in the PLU factorization. The multiplier is stored in the red color channel, and the reduced next column is stored in the blue color channel.

Listing A.2: Setting up row- and column-communicators

```
1 /* Set up row communicators */  
   MPI_Cart_sub(origcom, {0, 1}, &rowcom);  
  
   /* Set up column communicators */  
5 MPI_Cart_sub(origcom, {1, 0}, &colcom);
```

First, the current top row and pivot row are swapped, simultaneously as we eliminate using the multipliers in L . Because we are using two buffers, we read back the pivot row simultaneously using PBOs, and store them in main memory. When all rows in the block-row have been computed, U is sent to all nodes in the same column for the reduction operation.

Eliminate

The elimination procedure calculates multipliers. Normalized rows (U) are sent from the current time-step's pivot node, and the multipliers are calculated using these. The elimination step follows much of the same procedure as the pivot step, but it is a simpler case since there is no complications with row interchanges. This is again because the pivot node only pivots within in-core memory.

Reduce

The reduction step is trivial on the local node. Using a for-loop, we sequentially reduce the whole remaining sub-matrix by looking up one row from U and one column from L , and calculating the reduced A as $A_{i,j} := A_{i,j} - L_{k,s} \cdot U_{s,j}$.

Sending of data

This section describes how data is sent between different nodes. The use of MPI-2 for this inter-node communication will also be explained in detail.

Based on the algorithm discussed in Section A.3.1 we have the following communication scenarios:

1. Sending data to all processes in the same row as active process (to *normalize* and *reduce*).
2. Sending data to all processes in the same column as active process (to *eliminate* and *reduce*).

For broadcasting data to all processes in the same row as the active process, the broadcast function in MPI, `MPI_Bcast`, is used. This function takes a communicator, a pointer to the data, and a count of data elements as arguments. When called, it broadcasts the data to all processes within that communicator. Broadcasting data to the same row as yourself is done by calling `MPI_Bcast` with the row communicator.

To broadcast to columns we use the column communicator instead of the row communicator.

Since the `MPI_Bcast` function is collective, it needs to be called in every process within the current communicator. This implies that each process needs to know a priori from which node it will receive the next broadcast. In our application we

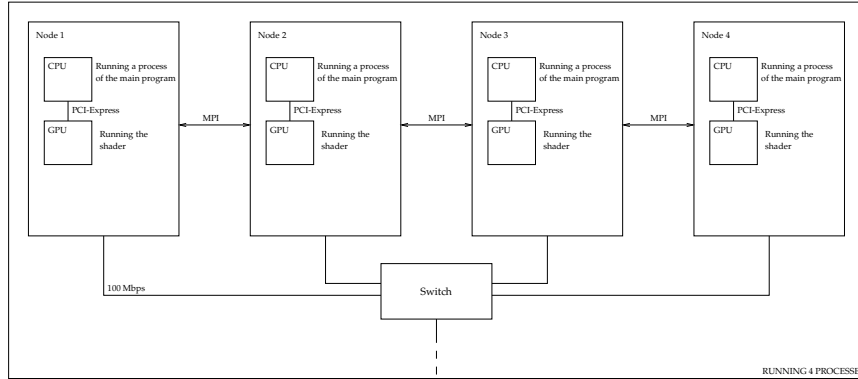


Figure A.4: Overview of physical setup of nodes.

have a function dedicated to calculate this. This function bases the calculation on which global pass the process is currently in, and which type it currently is (*pivot*, *normalize*, *eliminate* or *reduce*). This method is fairly complicated, but can be briefly explained as follows: The *normalize* nodes will always receive a broadcast from the *pivot* node, which is the diagonal element in its row communicator. *Eliminate* is similar, but will receive from the diagonal element in its column communicator. Finally, *reduce* will receive data from *normalize*, which is the node with the same column index as the current node, and the same row index as the current *pivot* node. *Reduce* also receives data from *eliminate*, which is computed in a similar fashion.

To facilitate the communication needed by our algorithm, row- and column-wise communicators are set up. Listing A.2 shows the code used to create these communicators. In this listing, the array sent as the second parameter sets which dimension we wish to keep in the new communicators. When we create the row communicators we keep the y-dimension intact, and when creating the column communicators we do the opposite and keep the x-dimension. When the code is executed, each process will set up a row communicator called `rowcom` and a column communicator called `colcom` relative to the process' location in the grid.

A.4 Results

The cluster which we benchmarked our application on consists of four one-CPU, one-GPU nodes as shown in Figure A.4. The nodes were all equipped with Intel Pentium 4 processors with Hyper-Threading Technology (HTT) and 2 GB of RAM. All nodes had an NVIDIA GeForce 7800 GT graphics adapter on a PCI-Express 16× slot.

A.4.1 Benchmark

Benchmarking of our algorithms showed that it gives sufficiently accurate results considering that all computation is executed on single precision hardware.

When benchmarking the algorithm, we have varied several variables to identify possible bottlenecks. The variables we have varied are:

1. Number of nodes.

Table A.1: Variation of the subsize parameter, as well as the impact of several nodes. The number of processes is 4, and the times are in seconds.

-		Nodes			
n	Subsize	1	2	3	4
128	8	0,20607	0,14006	0,13756	0,28482
	16	0,23247	0,11918	0,14593	0,28739
	32	0,19208	0,10213	0,11030	0,27609
	64	0,13572	0,09238	0,08232	0,24506
512	32	0,54457	0,25811	0,28454	1,11726
	64	0,49388	0,24161	0,26360	0,78500
	128	0,32307	0,23648	0,24194	0,64518
	256	0,24012	0,20242	0,21688	0,36138
2048	128	3,17257	2,43952	2,95311	3,19620
	256	3,07729	2,43028	2,95513	3,15248
	512	2,88925	2,41467	2,87859	3,05907
	1024	2,59612	2,39955	2,68849	2,93310
4096	256	13,76410	13,03520	14,77890	15,26550
	512	13,70820	13,18710	14,74090	15,78910
	1024	13,59550	13,59640	14,73430	16,26440
	2048	14,62520	14,45370	14,50600	16,66760

2. Number of processes.
3. The size of the block to factorize in each global pass (subsize).
4. The total size of the problem matrix (n).

In addition, we have benchmarked the pivot operation on a single node executed on the full matrix, as well as only network communication. This gives us performance results for our network setup, the local algorithm, as well as the global algorithm, enabling analysis of the limiting factor.

Table A.1 shows the time used to compute the PLU factorization while varying the number of nodes, size of the matrix, and the block size. The maximum achieved performance is 3.5 GFLOPS (for $n = 4096$ on two nodes), and the general trend seems to suggest that using only two nodes is faster than using four. This can somewhat be explained by intraprocess communication being faster with two processes per node, than one process per node, as this eliminates a lot of network communication.

Table A.2 shows the time used to compute the PLU factorization while varying the number of processes on four nodes. As the table shows, the speed of the algorithm can be greatly influenced by tuning this parameter. However, the optimal number of processes seems to vary with the size of the matrix. The maximum achieved performance achieved was now increased to 4.2 GFLOPS (16 processes on four nodes). We also timed the network-communication, and measured the percentage of the total time used for network communication. The percentages show that there is a substantial time used to send and receive data alone.

To analyze the impact of the network, we ran the network communication while varying the number of nodes. Table A.3 shows the time of the network

Table A.2: Variation of the number of processes. The number of nodes is four, and the times are in seconds.

Procs	Subsize	Time	Network time %
4	256	97,78950	42
	512	98,19350	36
	1024	99,65560	31
	2048	102,98800	30
16	256	86,30310	37
	512	88,69330	35
	1024	89,53110	35
	2048	95,1656	33
64	128	122,72900	24
	256	124,48000	23
	512	120,79000	23
	1024	124,32000	21

Table A.3: The time spent transmitting data. The number of processes is four and the problem size is 2048, while the number of nodes is varied. This shows the impact of the network communication.

-	Nodes			
Subsize	1	2	3	4
128	0,57228	3,18597	5,70082	6,30781
256	0,59500	3,14385	5,72201	5,73072
512	0,62175	3,13140	5,64543	5,65009
1024	0,69741	3,02938	5,37086	5,38025

Table A.4: The time spent computing using only a single node where subsize = n. The times are in seconds.

n	Time
64	0,0284489
256	0,0491337
1024	0,280545
2048	1,44955
4096	10,051

communication, and the impact of the subsize parameter, as well as the use of multiple nodes. The subsize parameter seems to have little effect on the time, whilst the number of nodes has a massive impact. Using two nodes with four processes is approximately half as expensive as using four nodes.

Finally, we have benchmarked the pivot operation on one node. This is the most computationally heavy operation, and a limiting factor. Table A.4 shows the time spent to compute a full matrix using the pivot operation. The peak performance was measured for the largest matrix, 4096×4096 , where the algorithm performed 4.6 GFLOPS. As a comparison, we timed the ATLAS implementation used in MATLAB, which achieved 3.5 GFLOPS on the same problem size.

A.4.2 Analysis

Our global algorithm had a maximum measured performance of 4.2 GFLOPS using four nodes, while our local algorithm showed a promising 4.6 GFLOPS. The network communication could account for at least 20% of the total runtime. However, because of the way the presented algorithm is executed, most of the processes simply idle, waiting for data. This is the largest bottleneck, but there are some solutions.

Using a look-ahead strategy, as used in the HPL [PWDC] algorithm, will increase the workload per node, but decrease the idling. In addition, restructuring the computation into smaller parts, so that pivot, eliminate, normalize and reduce are split into smaller subproblems, will also decrease the time spent idling per node.

We have not been able to show the full potential of this algorithm, because we have only have had four nodes at disposal. Having only four nodes makes almost all the computation execute serially, because we only have one node per operation at each global time-step. This parallelizes the computation of normalize and eliminate only. Using more nodes, will parallelize the reduction step of the algorithm as well, and probably speed up the total computational speed.

A.5 Conclusions and further research

We have presented a new way of computing the PLU factorization of a matrix, by using the GPU on a cluster of nodes. We have shown that the algorithms computed locally are efficient, and even outperforming ATLAS. Our global algorithm, however, is less efficient. We have pointed to a slow network link, a lot of idling of nodes, and the use of only four nodes as the main reasons.

A faster network link will decrease the impact of the network communication in our algorithm. It is also possible to lessen the issue with idling of nodes by using techniques such as look-ahead, or splitting up the computation further.

It is possible to extend our algorithm to include forward and backward substitution, as the HPL algorithm does. The computation of the forward substitution will be virtually free, while the backward substitution will require more global passes. Including the forward and backward substitution in the algorithm will fulfill the complexity demands for the Top500 benchmark [UUN].

A.6 Acknowledgements

We would like to thank J. Hjelmervik for first proposing this project to us, and our supervisors K.-A. Lie and T. R. Hagen for their helpful guiding and notes on our white paper. We would also like to thank G. W. Ma for running the communications tests on Simula's cluster, and our fellow master students at SINTEF ICT for insightful discussions.

Appendix B

Data Tables

The following tables contain times from the benchmarks of algorithms presented in this thesis, and they are meant as supplementary to the previously presented graphs.

Table B.1: Time used to compute the RREF of a matrix size $n \times (n + 1)$: (a) The internal implementation in MATLAB. (b) The GPU implementation compared to PLU factorization in MATLAB. Note that PLU factorization is a far less computationally demanding operation.

n	MATLAB (s)	n	MATLAB (s)	GPU (s)	Speedup
11	0.011455	138	0.0016	0.0286	0.0588
65	0.304077	266	0.0095	0.0440	0.2158
131	1.151880	388	0.0274	0.0666	0.4121
256	4.209507	520	0.0608	0.1070	0.5678
521	17.88167	646	0.1112	0.1534	0.7251
648	28.31852	776	0.1849	0.1936	0.9550
897	66.28932	906	0.2880	0.2816	1.0229
1029	97.1145	1028	0.4196	0.3770	1.1131
1285	181.1832	1280	0.7787	0.6727	1.1576
1543	293.8220	1536	1.3393	1.1371	1.1778
1798	453.8168	1796	2.0478	1.7786	1.1514
2051	647.5665	2055	3.0469	2.6496	1.1499

(a)

(b)

Table B.2: Time spent computing the matrix-matrix multiplication of an $m \times n$ and an $n \times o$ matrix: (a) The internal implementation in MATLAB. (b) The single-pass GPU algorithm. (c) The multi-pass GPU algorithm.

$\sqrt{\frac{mn+no}{c}}$	MATLAB (s)	$\sqrt{\frac{mn+no}{c}}$	GPU single-pass (s)
58.775	0.00024419	651.12	0.15053
121.68	0.0015211	685.91	0.13185
197.9	0.005967	982.22	0.37169
302.02	0.022959	1628	1.7318
357.22	0.036471	1653.2	1.8181
951.18	0.67207	2004.7	3.1735
1643.2	3.4754	2029.7	3.2999
2547.2	12.996	2502	6.1705
3049.2	22.6	2555.7	6.5888
3550.2	35.772	3003.5	10.72
4041.2	52.712	3086.2	11.672

(a)

(b)

$\sqrt{\frac{mn+no}{c}}$	GPU multi-pass (s)
51.303	0.0030236
70.887	0.0033631
138.74	0.0050918
262.93	0.012401
397.75	0.035368
520.99	0.051389
774.75	0.15116
1294.7	0.65857
2553	4.8447
3058.7	8.1347
3629.7	13.683
3961.2	17.624

(c)

Table B.3: Time of MATLAB and the GPU toolbox to compute the PLU factorization of a matrix of size $n \times n$.

n	MATLAB (s)	GPU (s)	Speedup
1024	0.4285	0.2969	1.4430
1208	0.6387	0.4053	1.5759
1261	0.7352	0.4466	1.6462
1290	0.7841	0.4702	1.6674
1644	1.5870	1.0021	1.5837
1696	1.7152	1.0944	1.5672
1696	1.7218	1.0944	1.5732
2002	2.7876	1.4241	1.9575
2058	3.0314	1.8741	1.6176
2062	3.0430	1.5393	1.9769
2083	3.1292	1.5856	1.9736
2091	3.1627	1.8937	1.6702
2503	5.3744	2.6168	2.0538
2554	5.6714	2.8308	2.0035
2590	5.8868	3.0886	1.9060
3002	9.0903	5.0169	1.8119
3089	9.9035	6.0506	1.6368
3093	10.018	6.0549	1.6545
3506	14.374	9.1683	1.5678
3539	14.771	8.7589	1.6864
3586	15.375	10.738	1.4318
3595	15.449	10.766	1.4350

Appendix C

Shallow water equations source code

This appendix contains parts of source code used to solve the shallow water equations using the ADI scheme described in Chapter 5. The parts shown are the ones which set up the system of linear equations, and solve them. It should be emphasized that the source code shown here is instructional, and by no means tuned for efficiency.

Listing C.1: Part of the MATLAB reference implementation of the ADI scheme for solving the shallow water equations.

```

1 %Set up the systems of linear equations (Ax = b)
  for jj=[1:x]
    A = zeros(y, y);
    b = zeros(y, 1);
5
    %Start boundary condition
    A(1, 1) = 1 + mu*alpha(2, jj);
    A(1, 2) = -mu*alpha(2, jj);
    b(1) = z{this}(2, jj+1) ...
10     - rho*( ...
        beta(1, jj+1).*v{this}(2, jj+1) ...
        - beta(1, jj).*v{this}(2, jj)) ...
    - sigma*( ...
        alpha(2, jj).*Fu(2, jj) ...
15     - alpha(1, jj).*Fu(1, jj));

    for ii=[2:y-1] %Only internal points,
      %A
      A(ii, ii-1) = - mu* alpha(ii, jj);
20     A(ii, ii) = 1 + mu*(alpha(ii, jj) + alpha(ii+1, jj));
      A(ii, ii+1) = - mu* alpha(ii+1, jj);

```

```

        %b
        b(ii) = z{this}(ii+1, jj+1) ...
25         - rho*( ...
            beta(ii, jj+1).*v{this}(ii+1, jj+1) ...
            - beta(ii, jj).*v{this}(ii+1, jj)) ...
        - sigma*( ...
            alpha(ii+1, jj).*Fu(ii+1, jj) ...
30         - alpha(ii, jj).*Fu(ii, jj));
    end

    %End boundary condition
    A(y, y-1) = -mu*alpha(y, jj);
35    A(y, y) = 1 + mu*alpha(y, jj);
    b(y) = z{this}(y+1, jj+1) ...
        - rho*( ...
            beta(y, jj+1).*v{this}(y+1, jj+1) ...
            - beta(y, jj).*v{this}(y+1, jj)) ...
40    - sigma*( ...
            alpha(y+1, jj).*Fu(y+1, jj) ...
            - alpha(y, jj).*Fu(y, jj));

    %Solve, a. This is the statement that is replaced
45    %when using both the CPU and the GPU.
    z{next}([2:y+1], jj+1) = A \ b;

    %Set boundary conditions
    z{next}([1 y+2], jj+1) = z{next}([2 y+1], jj+1);
50 end

    %Set boundary conditions
    z{next}([2:y+1], [1 x+2]) = z{next}([2:y+1], [2 x+1]);

55    %Find u at internal points
    u{next}([1:y+1], [2:x+1]) = (Fu([1:y+1], [1:x])
        - (2*g*sigma)*( ...
            z{next}([2:y+2], [2:x+1]) ...
            - z{next}([1:y+1], [2:x+1]))) ...
60    ./ (1 + dt*gamma([1:y+1], [1:x]));

    %Set boundary conditions
    u{next}([1:y+1], [1 x+2]) = u{next}([1:y+1], [2 x+1]);

```

Bibliography

- [Adv06] Advanced Micro Devices Inc. ATI CTM guide, technical reference manual. Online; http://ati.de/companyinfo/researcher/documents/ATI_CTM_Guide.pdf, 2006. [accessed 2007-04-13].
- [Adv07] Advanced Micro Devices Inc. Radeon X1900 graphics technology - GPU specifications. Online; <http://ati.amd.com/products/RadeonX1900/specs.html>, 2007. [accessed 2007-04-27].
- [Arg] Argonne National Laboratory. Mpich2. Online; <http://www-unix.mcs.anl.gov/mpi/mpich2/>. [accessed 2007-04-18].
- [Bes04] D. Besedin. Discovering ddr2-533 potential: Results of memory module tests at the fsb frequency of 266 mhz. Online; <http://www.digit-life.com/articles2/ddr2-rmma/ddr2-rmma-fsb266.html>, 2004. [accessed 2007-04-29].
- [BFGS03] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. *ACM Trans. Graph.*, 22(3):917–924, 2003.
- [BFH⁺04] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.
- [Bly06] D. Blythe. The Direct3D 10 system. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Papers*, pages 724–734, New York, NY, USA, 2006. ACM Press.
- [Cas90] V. Casulli. Semi-implicit finite difference methods for the two-dimensional shallow water equations. *Journal of Computational Physics*, 86:56–74, 1990.
- [CES71] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.
- [CW87] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. In *STOC '87: Proceedings of the nineteenth annual ACM conference on Theory of computing*, pages 1–6, New York, NY, USA, 1987. ACM Press.

- [Dav07a] T. Davis. University of Florida sparse matrix collection. NA Digest, vol. 92, no. 42, October 16, 1994, NA Digest, vol. 96, no. 28, July 23, 1996, and NA Digest, vol. 97, no. 23, June 7, 1997, Online; <http://www.cise.ufl.edu/research/sparse/matrices>, 2007. [accessed 2007-04-01].
- [Dav07b] T. Davis. University of Florida sparse matrix collection. Online; <http://www.cise.ufl.edu/~davis/techreports/matrices.pdf>, 2007. Submitted to SIAM Journal on Matrix Analysis and Applications. [accessed 2007-04-01].
- [DD06] G. Da Graça and D. Defour. Implementation of float-float operators on graphics hardware, 2006.
- [FSC06] D. Fay, A. Sazegari, and D. Connors. A detailed study of the numerical accuracy of GPU-implemented math functions. Online; http://www.gpgpu.org/sc2006/workshop/Apple_GPUintrinsic.pdf, 2006. [accessed 2007-04-23].
- [FSH04] K. Fatahalian, J. Sugerman, and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In *HWWS '04: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 133–137, New York, NY, USA, 2004. ACM Press.
- [GGHM05] N. Galoppo, N. K. Govindaraju, M. Henson, and D. Manocha. LU-GPU: Efficient algorithms for solving dense linear systems on graphics hardware. In *SC '05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing*, page 3, Washington, DC, USA, 2005. IEEE Computer Society.
- [GLGM06] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha. A memory model for scientific algorithms on graphics processors. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 89, New York, NY, USA, 2006. ACM Press.
- [GPG06] GPGPU. FP rounding — www.gpgpu.org/forum. Online; <http://www.gpgpu.org/forums/viewtopic.php?p=12636>, 2006. [accessed 2007-04-11].
- [GST05] D. Göddeke, R. Strzodka, and S. Turek. Accelerating double precision FEM simulations with GPUs. In Frank Hülsemann, Markus Kowarschik, and Ulrich Rüde, editors, *Proceedings of the 18th Symposium on Simulation Technique (ASIM 2005)*, pages 139–144. SCS Publishing House e.V, September 2005.
- [GST07] D. Göddeke, R. Strzodka, and S. Turek. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *International Journal of Parallel, Emergent and Distributed Systems*, 2007. accepted for publication November 2006, to appear.
- [HCH03] J. D. Hall, N. A. Carr, and J. C. Hart. Cache and bandwidth aware matrix multiplication on the GPU, 2003.
- [Hig02] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition, 2002.

- [Hof05] H. P. Hofstee. Introduction to the Cell Broadband Engine. Online; [http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D/\\$file/2053_IBM_CellIntro.pdf](http://www-306.ibm.com/chips/techlib/techlib.nsf/techdocs/D21E662845B95D4F872570AB0055404D/$file/2053_IBM_CellIntro.pdf), 2005. [accessed 2007-04-29].
- [JS05] C. Jiang and M. Snir. Automatic tuning matrix multiplication performance on graphics hardware. In *PACT '05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 185–196, Washington, DC, USA, 2005. IEEE Computer Society.
- [KAN05] KANDU and contributors. The gathering 2007 – still puzzled? Online; <http://www.gathering.org/tg07/>, 2005. [accessed 2007-05-01].
- [KBR06] J. Kessenich, D. Baldwin, and R. Rost. The OpenGL shading language. Online; <http://www.opengl.org/registry/doc/GLSLangSpec.Full1.1.20.8.pdf>, 2006. [accessed 2007-04-27].
- [Kem] W. E. Kempf. Class condition. Online; <http://www.boost.org/doc/html/condition.html>. [accessed 2007-04-09].
- [Kem05a] W. E. Kempf. Chapter 12. Boost.Threads. Online; <http://www.boost.org/doc/html/threads.html>, 2005. [accessed 2007-04-09].
- [Kem05b] W. E. Kempf. Mutexes. Online; <http://www.boost.org/doc/html/threads/concepts.html>, 2005. [accessed 2007-04-09].
- [Kho04] F. Khoury. History of the chézy formula. Online; <http://chezy.sdsu.edu/>, 2004. [accessed 2007-04-19].
- [Khr07] Khronos Group. OpenGL – the industry’s foundation for high performance graphics. Online; <http://www.opengl.org>, 2007. [accessed 2007-04-25].
- [Kni04] O. Knill. Use of linear algebra. Online; http://www.math.harvard.edu/archive/21b_fall_04/handouts/00-use.pdf, 2004. [accessed 2007-04-25].
- [KW03] J. Krüger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Trans. Graph.*, 22(3):908–916, 2003.
- [Käl07] A. Källander. Multithreading in MEX files. Email communication with Anna Källander, Technical Support Engineer, The MathWorks, 2007. [received 2007-02-08].
- [LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, 1979.
- [LKHW04] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. A streaming narrow-band algorithm: Interactive deformation and visualization of level sets. *IEEE Transactions on Visualization and Computer Graphics*, 10(40):422–433, July 2004.
- [LM01] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 55–55, New York, NY, USA, 2001. ACM Press.

- [LM04] A. N. Langville and C. D. Meyer. The use of the linear algebra by web search engines. 2004.
- [LSK⁺06] A. E. Lefohn, S. Sengupta, J. Kniss, R. Strzodka, and J. D. Owens. Glift: Generic data structures for the gpu. In *Proceedings of the 2006 Workshop on Edge Computing Using New Commodity Architectures*, pages D–15–16, May 2006.
- [LT93] N. G. Leveson and C. S. Turner. Investigation of the Therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.
- [Lyc06] T. Lyche. Lecture notes for INF-MAT 3350 / 4350, 2006. Lecture Notes for Numerical Linear Algebra course at the University of Oslo.
- [MD06] M. D. McCool and B. D’Amora. M08—programming using RapidMind on the Cell BE. In *SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 222, New York, NY, USA, 2006. ACM Press.
- [MDP⁺04] M. D. McCool, S. Du Toit, T. Popa, B. Chan, and K. Moule. Shader algebra. *ACM Trans. Graph.*, 23(3):787–795, 2004.
- [Mey04] C. D. Meyer. *Matrix Analysis and Applied Linear Algebra*. SIAM, 2004.
- [Mic04] Microsoft Corporation. PCI Express FAQ for graphics. Online; http://www.microsoft.com/whdc/device/display/PCIe_graphics.msp, 2004. [accessed 2007-04-17].
- [Mic07a] Microsoft Corporation. Common-shader core (Direct3D 10). Online; <http://msdn2.microsoft.com/en-us/library/bb205137.aspx>, 2007. [accessed 2007-04-25].
- [Mic07b] Microsoft Corporation. Microsoft DirectX. Online; <http://www.microsoft.com/directx>, 2007. [accessed 2007-04-25].
- [Mic07c] Microsoft Corporation. Shaders (Direct3D 10). Online; <http://msdn2.microsoft.com/en-us/library/bb205134.aspx>, 2007. [accessed 2007-04-27].
- [Mol00] C. Moler. MATLAB news & notes - winter 2000. Online; http://www.mathworks.com/company/newsletters/news_notes/clevescorner/winter2000.cleve.html, 2000. [accessed 2006-04-01].
- [Mor03] A. Moravánszky. Dense matrix algebra on the GPU. Online; <http://www.shaderx2.com/shaderx.pdf>, 2003. [accessed 2006-05-11].
- [MPI] MPI Forum. Mpi documents. Online; <http://www.mpi-forum.org/docs/docs.html>. [accessed 2007-04-18].
- [MWHL06] M. McCool, K. Wadleigh, B. Henderson, and H.-Y. Lin. Performance evaluation of GPUs using the RapidMind development platform. Online; <http://rapidmind.net/pdfs/RapidMindGPU.pdf>, 2006. [accessed 2007-04-25].
- [MY02] I. Mcleod and H. Yu. Timing comparisons of Mathematica, MATLAB, R, S-Plus, C & Fortran. Online; <http://www.stats.uwo.ca/faculty/aim/epubs/MatrixInverseTiming/default.htm>, 2002. [accessed 2007-04-02].

- [Neo07] Neoptica. Programmable graphics – the future of interactive rendering. Online; <http://www.neoptica.com/NeopticaWhitepaper.pdf>, 2007. [accessed 2007-04-23].
- [NVI05] NVIDIA Corporation. NVIDIA GPU programming guide. Online; http://developer.nvidia.com/object/gpu_programming_guide.html, 2005. Version 2.4.0 [accessed 2007-04-17].
- [NVI07a] NVIDIA Corporation. Cg. Online; http://developer.nvidia.com/page/cg_main.html, 2007. [accessed 2007-04-26].
- [NVI07b] NVIDIA Corporation. CUDA programming guide version 0.8.1. Online; http://developer.download.nvidia.com/compute/cuda/0.81/NVIDIA_CUDA_Programming_Guide_0.8.1.pdf, 2007. [accessed 2007-04-23].
- [NVI07c] NVIDIA Corporation. Geforce 8800. Online; http://www.nvidia.com/page/geforce_8800.html, 2007. [accessed 2007-04-27].
- [OLG⁺05] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.
- [OLG⁺07] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.
- [Pea06] PeakStream. The PeakStream platform: High productivity software development for multi-core processors. Online; http://peakstreaminc.com/reference/peakstream_platform_technote.pdf, 2006. [accessed 2007-04-23].
- [PWDC] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. Hpl. <http://www.netlib.org/benchmark/hpl/>. [accessed 2007-04-18].
- [Rob05] S. Robinson. Toward an optimal algorithm for matrix multiplication. *SIAM News*, 38(9), November 2005.
- [SDK05] R. Strzodka, M. Doggett, and A. Kolb. Scientific computation for simulations on programmable graphics hardware. In *Special Issue: Programmable Graphics Hardware*, volume 13, 2005.
- [SIN06] SINTEF ICT. GPGPU - graphics hardware as a high-end computational resource. Online; <http://www.sintef.no/gpgpu>, 2006. [accessed 2007-04-17].
- [ST04] R. Strzodka and A. Telea. Generalized distance transforms and skeletons in graphics hardware. In *VisSym*, pages 221–230, 2004.
- [Str02] R. Strzodka. Virtual 16 bit precise operations on rgba8 textures, 2002.
- [SWND05] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL Programming Guide, Fifth Edition, The Official Guide to Learning OpenGL, Version 2*. Addison-Wesley, 2005.
- [The06] The MathWorks. MATLAB software acknowledgements. Online; <http://www.mathworks.com/access/helpdesk/help/base/relnotes/software.html>, 2006. [accessed 2006-05-14].

- [TSV05] B. Töpelt, D. Schuhmann, and F. Völkel. The mother of all CPU charts 2005/2006. Online; http://www.tomshardware.com/2005/11/21/the_mother_of_all_cpu_charts_2005/, 2005. [accessed 2007-03-31].
- [UUN] University of Mannheim, University of Tennessee, and NERSC/LBNL. Top 500 supercomputer sites. <http://www.top500.org/about>. [accessed 2007-04-18].
- [WD98a] J. Wasñniewski and J. J. Dongarra. High performance linear algebra package LAPACK90. Technical Report UT-CS-98-384, 1998.
- [WD98b] R. C. Whaley and J. J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.
- [Wik07a] Wikipedia. Floating point — wikipedia, the free encyclopedia. Online; http://en.wikipedia.org/w/index.php?title=Floating_point&oldid=122873488, 2007. [accessed 2007-04-17].
- [Wik07b] Wikipedia. Moore's law — wikipedia, the free encyclopedia. Online; http://en.wikipedia.org/w/index.php?title=Moore%27s_Law&oldid=118386303, 2007. [accessed 2007-03-31].
- [Wik07c] Wikipedia. Thread (computer science) — wikipedia, the free encyclopedia. Online; http://en.wikipedia.org/w/index.php?title=Thread_%28computer_science%29&oldid=121334515, 2007. [accessed 2007-04-09].
- [Wik07d] Wikipedia. Thread-safe — wikipedia, the free encyclopedia. Online; <http://en.wikipedia.org/w/index.php?title=Thread-safe&oldid=114790643>, 2007. [accessed 2007-04-09].